The Probabilistic Model Checker Storm – Symbolic Methods for Probabilistic Model Checking

Christian Hensel

Department of Computer Science

Technical Report

Aachener Informatik-Berichte (AIB) | ISSN 0935-3232 | AIB-2018-06
RWTH Aachen University | Department of Computer Science | December 2018

The Probabilistic Model Checker Storm
Symbolic Methods for Probabilistic Model Checking

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften
der RWTH Aachen University zur Erlangung des akademischen
Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Dipl.-Inform. Christian Hensel

aus

Leipzig

Berichter: Prof. Dr. Ir. Joost-Pieter Katoen
Dr. David Parker

Tag der mündlichen Prüfung: 3.12.2018

Abstract

In a world in which we increasingly rely on safety critical systems that simultaneously are becoming ever more complex, formal methods provide a means to mathematically rigorously prove systems correct. Model checking is a fully automated technique that is successfully applied in the verification of software and hardware circuits. However, in practice many systems exhibit stochastic behavior, for instance when components may fail or randomization is used as a key element to improve efficiency. Probabilistic model checking extends traditional (qualitative) model checking to deal with such systems. As model checking can be (simplistically) viewed as an exhaustive exploration of the state space of the model under consideration, it suffers from the curse of dimensionality: State spaces grow exponentially in the number of components and variables and they quickly become too large to be effectively manageable, a problem that is typically referred to as state space explosion.

Symbolic methods have helped to alleviate this problem substantially. Rather than considering states and transitions of the system individually, they instead exploit structure in the model and treat *sets* of states and transitions simultaneously. Model checkers based on symbolic techniques dominate the landscape of hardware and software model checking. In the probabilistic setting, symbolic methods too show potential but are arguably not on par with their qualitative counterparts. This thesis is concerned with advances in the field of symbolic techniques in the context of probabilistic model checking. The major contributions are fivefold:

(i) We propose the **JANI** modeling language to unify the multitude of input languages of probabilistic model checkers. It covers a large range of models involving randomization and timing aspects and offers well-defined points for future extensions.

(ii) We show how counterexamples on the level of **JANI** specifications can be synthesized. For this, we develop a method based on the connection of standard probabilistic model checking and a smart enumeration of the solutions of a satisfiability problem.

(iii) We combine the strengths of decision diagrams for the representation of gigantic systems and bisimulation minimization, a technique that reduces systems by factoring out symmetry. Our implementation is shown to drastically reduce the sizes of models involving probabilities, continuous time, nondeterminism and rewards.

(iv) We summarize, extend and implement a game-based abstraction-refinement framework that is able to treat infinite-state probabilistic automata. In contrast to other implementations, ours is openly available and computes strictly sound bounds through the use of rational arithmetic.

(v) We present Storm, a new high-performance probabilistic model checker. It goes beyond the standard verification tasks through numerous features and in particular integrates the items (i) to (iv) above. We show that it outperforms other state-of-the-art model checkers on the majority of instances of the Prism benchmark suite.

Zusammenfassung

In einer Welt, in der wir zusehends von sicherheitskritischen Systemen abhängen, die zeitgleich immer komplexer werden, bieten formale Methoden eine Möglichkeit, die Korrektheit von Systemen mathematisch rigoros zu beweisen. Model Checking ist eine vollautomatisierte Technik, die bereits erfolgreich zur Verifikation von Soft- und Hardware eingesetzt wird. Interessante Systeme weisen oft stochastisches Verhalten auf, beispielsweise wenn Komponenten potentiell ausfallen oder Randomisierung als Schlüssel zur Effizienzsteigerung benutzt wird. Probabilistisches Model Checking erweitert das traditionelle (qualitative) Model Checking um die Analyse solcher Systeme. Da Model Checking vereinfacht als vollständige Untersuchung des Zustandsraums des betreffenden Modells betrachtet werden kann, leidet es am Fluch der Dimensionalität: Zustandsräume wachsen exponentiell in der Anzahl der Komponenten und Variablen und werden schnell zu groß für eine Analyse — ein Problem, das auch Zustandsraumexplosion genannt wird.

Symbolische Methoden haben entscheidend dazu beigetragen, diesen Effekt abzuschwächen. Anstatt Zustände und Übergänge des Systems separat zu betrachten, versuchen sie, die Struktur des Modells auszunutzen und *Mengen* von Zuständen und Transitionen gleichzeitig zu behandeln. Im Bereich des traditionellen Model Checking dominieren Werkzeuge, die auf symbolischen Methoden basieren. Im probabilistischen Kontext zeigen symbolische Methoden ebenfalls Potential, sind aber in der Regel nicht auf Augenhöhe mit ihren qualitativen Gegenstücken. Diese Arbeit beschäftigt sich mit Fortschritten auf dem Gebiet der symbolischen Methoden im probabilistischen Model Checking. Die fünf wesentlichen Beiträge sind die Folgenden:

(i) Wir schlagen die **JANI**-Modellierungssprache vor, um die Vielzahl verschiedener Eingabesprachen von probabilistischen Model Checkern zu vereinheitlichen. Sie deckt eine große Bandbreite von Modellen ab, die Randomisierung oder Zeitaspekte beinhalten, und bietet wohldefinierte Stellen für zukünftige Erweiterungen.

(ii) Wir zeigen, wie Gegenbeispiele auf der Ebene von **JANI**-Spezifikationen synthetisiert werden können. Dazu kombinieren wir gewöhnliches probabilistisches Model Checking mit einer geschickten Aufzählung der Lösungen eines Erfüllbarkeitsproblems.

(iii) Wir vereinen die Stärken von Entscheidungsdiagrammen zur Darstellung von riesigen Systemen und Bisimulationsminimierung, die Zustandsräume durch die Faktorisierung von Symmetrien reduziert. Unsere Experimente zeigen, dass Modelle, die Wahrscheinlichkeiten, kontinuierliche Zeit und Kosten umfassen, drastisch verkleinert werden können.

(iv) Wir resümieren, erweitern und implementieren ein Framework zur spielbasierten Abstraktion, das unendliche probabilistische Automaten behandeln kann. Im Gegensatz zu anderen Implementierungen ist unsere öffentlich verfügbar und berechnet korrekte Schranken durch die Benutzung rationaler Arithmetik.

(v) Wir präsentieren STORM, einen neuen, hochperformanten probabilistischen Model Checker, der zahlreiche Funktionen bietet, die über die gewöhnlichen Verifikationsaufgaben hinausgehen, und insbesondere (i) bis (iv) vereint. Wir zeigen, dass STORM viele Instanzen der PRISM-Benchmarksammlung schneller löst als konkurrierende Werkzeuge.

The most important things are the hardest to say,
because words diminish them.

Stephen King

Acknowledgements

More than ten years ago, Benedikt pointed out open student assistant positions at the chair of computer science for the modeling and verification of software. The advertisement had a rocket on it and who does not want to do rocket science? Long story short, we applied, Thomas (N), and Viet Yen interviewed us both and we were thrilled to be hired shortly after. Since that day, I am hooked to this research group: I wrote my Diploma thesis at the chair and when Joost-Pieter gave me the opportunity to do my PhD in his group, I did not have to think long about it. It has been quite a journey with numerous ups and downs and I am proud that I was part of this group through the good times and the bad.

After all those years, I will probably forget to mention people that shared part of my journey and would definitely belong here. In wise anticipation, I already apologize for forgetting to include someone. So where do I start? As Lewis Carroll suggested, I will begin at the beginning and work my way to the very end. I want to thank Benedikt, Viet Yen, and Thomas for introducing me to the world of model checking and it's very hard to imagine where I would have ended up without you guys.

Thank you, Joost-Pieter, for giving me the opportunity to become a member of your research group. Thank you for all your advice, thank you for giving me so much freedom but at the same time being there whenever needed. Thank you for having my back, be it in the communication challenges of the CARP project or after raising a few (legitimate) questions (a) through (g) via mail.

It was Joost-Pieter who suggested that I visit Oxford for two months when writing my Diploma thesis. And it was thanks to Dave and Marta Kwiatkowska to make that possible. Thanks for having me, I enjoyed this wonderful city and am proud to say that I represented the university of Oxford in a table tennis match (against Warwick, I believe). Working with the creator of the most well-known tool in the field humbled and motivated me. Thank you, Dave, for reviewing this thesis and making it "PRISM"-approved. Thanks Mark (Timmer) and Henri (Hansen) for the memorable road trips through the UK and walks through Port Meadow.

Quite naturally, I have had the joy of working with several generations of colleagues. I want to thank Viet Yen, Martin, Carsten, and Jonathan for giving me a warm welcome at the chair when I was still a student assistant. Thank you Viet Yen, for the life lessons you taught Benedikt and me during our numerous meetings. And thank you, Jonathan, for still including us in the "TTipp" every year!

Even though I had a hard time wrapping my head around his probabilistic programs, I enjoyed sharing the office (and once even a king-sized bed) with Friedrich. Friedrich, I still maintain that Walter was perfectly reasonable to start cooking meth though. Thank you Friedrich and Christina for sharing the CARP experience™, several travels and pub visits. Christina, I am sorry for blabbering so much with you on that train ride in North Italy until that old guy snapped at us and left.

Thank you, Nils, for our fruitful collaboration, joint trips, sharing rooms at Kleinwalsertal and the occasional visit in a Shisha bar in Freiburg.

As I am probably the talkative kind of guy, I regularly dropped by several offices and either got questions (or rants) off my chest or simply started a programming languages discussion (I know, it's very nerdy). Thanks for always having an open door for me Harold, Tim (L), Tim (Q), Christoph, Florian, Matthias, Gereon, Stefan, Rebecca, and Marcel, I enjoyed all of those unnecessary discussions and you broadened my horizon. Thank you, Harold, for solving all technical problems there were, are or ever will be.

Speaking about talking a lot, I want to thank my current office mate Sebastian for all the laughs we shared. Keep up your can-do attitude with your overall energetic approach and the probabilistic model checking world will tremble.

Matthias, I know I have been hard on you during our various foosball matches, but I swear that this was just to bring out the best in you. I enjoyed all those moments and it was an honor to play with you, my captain.

Tim (Q), I still recall our first discussion atop E3 over pizza and beer. Despite my advertisement you became a member of our chair and went on to teach me more about uniqueness of fixpoints, thank you for that!

Dear Stormers (Sebastian, Matthias, Tim): I thank you for your dedication and the countless hours spent on Storm. Without you, it would not be what it is right now. Oh and also thanks for proof-reading individual chapters!

Also thanks to the students contributing to Storm be it as assistants or working on their theses, especially Dimitri and Philipp.

Joachim, thanks for contributing to Storm as well, for the countless chats and, in particular, for stressing me out and calming me down before the defense.

I want to thank all the colleagues with whom I have been co-organizing teaching courses. Thank you Jonathan, Viet Yen, Thomas (S), Friedrich, Harold, Christina, Sebastian, Matthias, and Benni. As you know, teaching is very dear to me and without you, it would not have been half the fun.

Thanks to everyone that enriched the many trips that I was lucky enough to conduct. In particular, thanks, Dennis, for joining me for a trip along the Australian east coast after ATVA 2014, one of the most adventurous and memorable trips of my whole life.

Let me return to my roots. I want to thank Benedikt for giving me a strong incentive to switch universities, another piece of the butterfly effect puzzle. Thank you, Pavel, Bene, and Marco for all those numerous beers that made student life and solving exercise sheets so much more enjoyable. In the same vein, I want to thank Angie for philosophical meetings next to the Rhein and always being there when I needed you. I want to thank Kadda and Fabian for taking my mind off of computer science once in a while and also for actually bothering to have a (very motivated) look into a draft of this document.

Lastly, I want to thank my family. Thank you, mum and dad, for your love and support over the last 32 years. I still recall (though vaguely) how my father bought me the Turbo Pascal compiler, a seemingly insignificant event that turned out to be an important building block towards my PhD. I want to thank my great-cousin Jens for fostering my interest in programming, providing me with access to Visual Basic and even giving me the chance to be a “professional” programmer for his company when I was 14. My late great-uncle Hans-Jürgen not only supported me financially, he also strongly encouraged me to follow the path he took and do a PhD. It greatly saddens me that you cannot see that I made it.

I want to thank my wife Jera with all my heart. You are the only one who really experienced all my ups and downs, all my enthusiasm and desperation during those last years. You were the one who gave me constant and endless support for which I am deeply grateful and there certainly is a chance that I would not have completed my PhD without you. Our son Julius and you are the dearest things to me. Who would be surprised that we met at the chair?

Contents

Chapter 1

Introduction

In an age of information, digital systems are everywhere. We are surrounded by them every day of our lives in the form of smartphones, transportation, medical devices and even kitchen aids. *And* we are relying on these systems to work correctly[1]. While in many cases failures are not critical, they may have catastrophic consequences in others. In the best of cases, this may be the loss of millions of dollars [Pra95; Koc+18; Lip+18], in other cases it may involve the loss of lives [Lev17].

At the same time and somewhat alarmingly, the systems we depend on become more and more complex. It is not rare to find more than fifty communicating miniature computers in modern cars that are concerned with various safety-critical tasks such as ABS, traction control, electronic stability control and more recent ones like collision avoidance systems or drive-by-wire. Apart from security concerns introduced by these cars' software being connected to the internet increasingly often, failures in these systems may have disastrous implications.

With the recent advent of methods based on artificial intelligence (AI) and machine learning, the problem becomes even more troublesome. Computers deal with millions of dollars in a matter of milliseconds by autonomous stock trading decisions[2] or detecting fraud. Airplanes can operate fully autonomously since decades and nowadays "ultimately, if required, [...] initiate an automatically flown evasive maneuver" [Wus02] thus possibly overriding the pilots if they don't respond to warnings within a certain amount of time. The trend towards autonomous vehicles on public streets is — for the better or worse —

[1] `https://xkcd.com/2030/`
[2] `https://www.bbc.com/news/magazine-19214294`

probably irreversible. The correct functioning of these systems is nothing short than a matter of life and death[3].

Clearly, this substantiates the need for techniques that guarantee the correctness of hardware and software systems with respect to their specification. In order to increase the confidence in the system design, *testing* efforts constitute a major part in any engineering project. The system in question is exposed to a number of inputs and its behaviors under these inputs is then compared with the expected behavior. If a mismatch is found, the system design needs to be revised. Testing can be partly automated and has the fundamental advantage that it can be applied to virtually all systems. However, ultimately, testing can only show the presence of erroneous behavior and not prove its absence, since, in practice, it is infeasible to exhaustively test all possible program inputs or system executions.

Research [Ber+18] shows that real-world requirements contain ambiguous wording, inconsistencies and underspecification. *Formal methods* try improving on the situation by requiring the system and the specification to be formalized in an unambiguous way. They arrive at a univocal analysis result by checking the formal model against the formal specification using mathematically rigorous reasoning. In fact, many safety and security standards such as ITSEC [Jah91; Woo+08], Common Criteria[4] [Ric+04], DO-178C and DO-278A [GP12], ISO 26262 [ISO11a] and IEC 62279 [IEC15] strongly recommend or require the application of formal methods in varying degrees. Formal methods have proven to find fundamental problems that have been hidden for years [Gou+15].

One of the most popular formal methods is *model checking*. It is a push-button technique that proves or refutes a (formal) property on a (formal) model of the system. Very briefly speaking, model checking amounts to an exhaustive search through the state space. After all, only if all possible evolutions of a system have been explored, it can be determined whether the specification is actually met. Figure 1.1 gives an overview of the model checking approach. A model checker takes the formal system model and the formal property as inputs and, somewhat simplifying, returns one of three results. It can report that the property holds or is violated. In the latter case, counterexamples can be synthesized that explain the cause of the violation as succinctly as possible, an effective means to convince engineers of a design problem. The third outcome is that the model checker ran out of resources. That is, it may take too long or too much memory to perform model checking. Responsible for this is the most well-known challenge in model checking: the state space explosion problem. As the state space may grow exponentially in the number of components of the system, "naive" model checking

[3]https://www.ntsb.gov/investigations/AccidentReports/Pages/HWY18FH011-preliminary.aspx
[4]`https://www.commoncriteriaportal.org/`

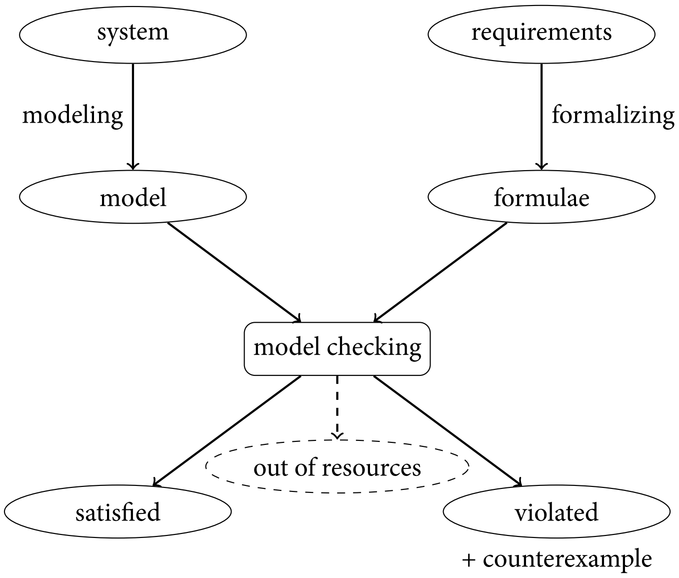


Figure 1.1: Overview of the model checking approach [BK08].

quickly becomes infeasible. Unsurprisingly, a (if not *the*) major part of research on model checking is dedicated to improve its scalability, for instance by applying smart representations and abstractions.

Model checking has written numerous success stories [BLR11; Hol14]. In fact, model checking is so successful that its inventors Edmund M. Clarke, E. Allen Emerson and Joseph Sifakis were awarded the Turing Award in 2007, the highest distinction in computer science that is recognized as the "Nobel Prize of computing".

Probabilistic model checking extends traditional model checking with tools and techniques for the analysis of systems involving random phenomena or other forms of behavior that can be approximated by randomization. Distributed algorithms and communication protocols are natural examples for this, since they often use randomization to efficiently break symmetry. Another example are cyber-physical systems that tightly integrate software and hardware such as sensors, actors and micro-controllers. Now, for instance, sensor readings may be noisy, actors may not always have the same effects and physical components may fail. Other domains that give rise to models involving

involving probabilistic aspects include, e. g. security protocols and systems biology.

In the context of probabilistic model checking, the state space explosion problem becomes even more pressing. Current techniques mostly require an in-memory representation of the full system under consideration. At the same time, the numerical solution techniques that are at the heart of probabilistic model checking become substantially more computationally expensive as the model grows. In 2008, in a retrospective on 25 years of model checking [Cla08], Edmund Clarke states that probabilistic model checking is an important challenge for the future that "require[s] major breakthroughs in order to become sufficiently practical for widespread use in industry."

One step in this direction are the application of *symbolic methods* in probabilistic model checking. Under this umbrella term, we understand all the techniques that try to avoid processing individual states but rather reason at the level of *sets* of states and *sets* of transitions. This starts at the level of the model specification: Instead of an explicit enumeration of states, the states (and behavior) of the model is rather specified symbolically much similar to a program in a programming language. Then, instead of building an explicit in-memory representation, the system may be represented (and analyzed) using data structures that efficiently capture the structure of the model. Taking this one step further, techniques may try to pursue an abstraction-refinement approach that initially does not distinguish any states of the system and only refines the view as necessary. These symbolic methods are the topic of this thesis.

1.1 Contributions and Outline

The individual chapters of the thesis can be read mostly independently. Only Chapters 2 and 3 are strictly required to understand the remaining Chapters 4 to 6. We now give an outline of this thesis and summarize the core contributions along the way.

After this introductory chapter, Chapter 2 paves the way for the remaining chapters: We give notations, formal definitions of the considered probabilistic models and properties and conclude with background on symbolic data structures and solving techniques.

The JANI Modeling Language and its Semantics. We already hinted at the fact that symbolic methods in probabilistic model checking naturally require a symbolic specification of the input model. There exist several symbolic modeling languages for probabilistic models, but they are associated with certain limitations or disadvantages. Therefore, many tools either introduce their own ad-hoc language or accept a dialect of a previously existing language. This introduces artificial boundaries between tools and, as a consequence, has led to a severely scattered tool landscape. In an attempt to address the major issues relevant to language design for probabilistic verification, we developed **JANI** [Bud+17], an expressive language that allows to formulate models that involve randomized or timed behavior. The language conservatively extends existing formalisms, is easily parseable and has well-defined extension points to be flexible enough to deal with future developments. Based on [Bud+17], Chapter 3 introduces **JANI** but significantly goes beyond the publication by providing a formal semantics for the fragment that corresponds to Markov reward automata. All remaining chapters build on **JANI** and its semantics.

High-Level Counterexamples for JANI Models. Counterexamples play an important role in the acceptance of model checking among engineers. They provide succinct feedback and substantiate failed verification attempts. In the context of probabilistic model checking, high-level counterexamples were recently proposed [Wim+13; Wim+15] to lift the previously existing notions of (probabilistic) counterexamples from the state space level back to the symbolic specification in terms of a **PRISM** program. However, the computation of such counterexamples through the proposed MILP-based formulation proved to be prohibitively expensive for larger models. In [Deh+14], we therefore developed an algorithm that computes high-level counterexamples by leveraging off-the-shelf probabilistic model checking in conjunction with a smart enumeration of the solutions of a satisfiability problem. We lift this algorithm from **PRISM** to **JANI** specifications in Chapter 4. Through a thorough experimental evaluation, we show that we achieve speedups of several orders of magnitude over the MILP solution.

Symbolic Bisimulation Minimization. We have already mentioned the state space explosion problem as one the most fundamental challenges in model checking and how two of the main approaches to alleviate the problem are symbolic representations and abstraction. One of the most well-understood techniques falling in the latter category is bisimulation minimization. It is often able to significantly reduce the size of structured models and at the same time preserves *all* properties of the most common probabilistic logics. [Wim10] proposes to combine the advantages of bisimulation minimization with the symbolic representation of systems using decision diagrams. In Chapter 5, we show how this idea can be lifted to nondeterministic, probabilistic and randomly timed systems equipped with reward (or cost) structures. Furthermore, we illustrate how the reduced model can be extracted in an explicit representation after the procedure. We conclude with an extensive evaluation of the effectiveness of the presented technique and show that it yields substantial state space reductions.

Game-Based Abstraction-Refinement for Probabilistic Automata. Bisimulation minimization per se preserves all properties in some fragment of a suitable temporal logic. An alternative approach that is is very successful in traditional model checking is typically referred to as *abstraction-refinement*. It starts with an abstraction of the concrete model. By construction, the abstract model soundly over-approximates the concrete model and analyzing the abstraction therefore allows to draw conclusions about the original model. As long as the information obtained this way is not precise enough, the abstraction is refined. In contrast to bisimulation minimization, the refinement is driven by the (abstract) analysis result and a concrete property and hence is tailored towards a specific verification task. Using stochastic games as abstractions for probabilistic automata permits to derive lower and upper bounds on both extremal reachability probabilities in the original model. Based on a Master thesis [Boh14], Chapter 6 shows how to implement game-based abstraction refinement. We fully automatically extract abstract stochastic games from **JANI** specifications through the formulation as a satisfiability problem, analyze the abstractions and finally refine them as necessary. For this, the chapter summarizes previous results on game-based abstraction refinement for probabilistic automata [Wac11; Kat+10]. The contribution of this chapter is not the technique itself, but rather several optimizations and amendments to its details as well as the implementation in the context of Storm. Our open-source implementation is the only maintained one that is applicable to general infinite probabilistic automata and, in particular, it is the only one that can compute structurally sound bounds through the use of rational arithmetic and policy iteration.

The Storm model checker. Support for **JANI** and the techniques presented in Chapters 4 to 6 has been realized in a new probabilistic model checker Storm. Chapter 7 is based on [Deh+17] and is concerned with the characteristics of this new tool. After stating the development goals of Storm, we discuss its defining features and design decisions. We then extensively evaluate its performance on the well-known Prism benchmark suite. With the help of **JANI**, we compare Storm with three other state-of-the-art probabilistic model checkers Prism, Epmc and the Modest Toolset. Our experiments indicate that Storm performs favorably across all verification approaches and model classes.

1.2 Publications

We now briefly discuss which publications of the author contain contributions that are part of this thesis and which do not. In general, we only list peer-reviewed publications and — to avoid confusion — remark that they were published under my birth name.

Relevant Publications.

[Bud+17] Carlos E. Budde, Christian Dehnert, Ernst Moritz Hahn, Arnd Hartmanns, Sebastian Junges, and Andrea Turrini. “JANI: Quantitative Model and Tool Interaction”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*. Vol. 10206. 2017, pp. 151–168 (cit. on pp. 5, 9, 46, 48, 73).

[Deh+14] Christian Dehnert, Nils Jansen, Ralf Wimmer, Erika Ábrahám, and Joost-Pieter Katoen. “Fast Debugging of PRISM Models”. In: *Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings*. Vol. 8837. 2014, pp. 146–162 (cit. on pp. 5, 9).

[Deh+17] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. “A Storm is Coming: A Modern Probabilistic Model Checker”. In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*. Vol. 10427. 2017, pp. 592–600 (cit. on pp. 6, 9, 80).

Further Publications.

[Ábr+14] Erika Ábrahám, Bernd Becker, Christian Dehnert, Nils Jansen, Joost-Pieter Katoen, and Ralf Wimmer. “Counterexample Generation for Discrete-Time Markov Models: An Introductory Survey”. In: *Formal Methods for Executable Software Models - 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2014, Bertinoro, Italy, June 16-20, 2014, Advanced Lectures*. Vol. 8483. 2014, pp. 65–121.

[Deh+12] Christian Dehnert, Daniel Gebler, Michele Volpato, and David N. Jansen. “On Abstraction of Probabilistic Systems”. In: *Stochastic Model Checking. Rigorous Dependability Analysis Using Model Checking Techniques for Stochastic Systems - International Autumn School, ROCKS 2012, Vahrn, Italy, October 22-26, 2012, Advanced Lectures*. Vol. 8453. 2012, pp. 87–116.

[Deh+15] Christian Dehnert, Sebastian Junges, Nils Jansen, Florian Corzilius, Matthias Volk, Harold Bruintjes, Joost-Pieter Katoen, and Erika Ábrahám. "PROPhESY: A PRObabilistic ParamEter SYnthesis Tool". In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Vol. 9206. 2015, pp. 214–231 (cit. on pp. 163, 244, 248).

[Deh+16] Christian Dehnert, Sebastian Junges, Nils Jansen, Florian Corzilius, Matthias Volk, Joost-Pieter Katoen, Erika Ábrahám, and Harold Bruintjes. "Parameter Synthesis for Probabilistic Systems". In: *19th GI/ITG/GMM Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, MBMV 2016, Freiburg im Breisgau, Germany, March 1-2, 2016*. 2016, pp. 72–74.

[DKP13] Christian Dehnert, Joost-Pieter Katoen, and David Parker. "SMT-Based Bisimulation Minimisation of Markov Models". In: *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*. Vol. 7737. 2013, pp. 28–47 (cit. on p. 165).

[Jan+16] Nils Jansen, Christian Dehnert, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Lukas Westhofen. "Bounded Model Checking for Probabilistic Programs". In: *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*. Vol. 9938. 2016, pp. 68–85 (cit. on pp. 307, 309).

[Jun+16] Sebastian Junges, Nils Jansen, Christian Dehnert, Ufuk Topcu, and Joost-Pieter Katoen. "Safety-Constrained Reinforcement Learning for MDPs". In: *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Vol. 9636. 2016, pp. 130–146 (cit. on p. 245).

[Qua+15] Tim Quatmann, Nils Jansen, Christian Dehnert, Ralf Wimmer, Erika Ábrahám, Joost-Pieter Katoen, and Bernd Becker. "Counterexamples for Expected Rewards". In: *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*. Vol. 9109. 2015, pp. 435–452.

[Qua+16] Tim Quatmann, Christian Dehnert, Nils Jansen, Sebastian Junges, and Joost-Pieter Katoen. "Parameter Synthesis for Markov Models: Faster Than Ever". In: *Automated Technology for Verification and Analysis - 14th*

International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings. Vol. 9938. 2016, pp. 50–67 (cit. on p. 244).

Contributions of the Author to the Relevant Publications. All publications mentioned above are the result of the joint work of many people, countless meetings in offices or via teleconferences and an incredible amount of writing and programming hours. However, I am required to measure my contributions — at least concerning the relevant publications. In general, I helped writing or at least finalizing all publications and I therefore rather discuss the contributions concerning the development of ideas and the implementation aspects.

For [Bud+17], the main parts and design decisions of the **JANI** language had already been shaped when Sebastian Junges and I were asked whether we would like to join this research project. During a number of frequent Skype calls with all authors, we developed the initial proposal substantially further. Concerning the **JANI**-support within Storm, I implemented the explicit and symbolic model generation showcased in [Bud+17] and the conversion of **PRISM** models to **JANI** that is also mentioned there.

For [Deh+14], the fundamental idea to leverage a MaxSat solver came from Ralf Wimmer and Nils Jansen. I implemented a prototype version using a MaxSat-based approach and, for reference, the Milp-based technique proposed in [Wim+15]. I soon realized that the performance was unsatisfactory and that crucial optimizations were missing. I therefore experimented with faster MaxSat solution techniques, developed sharper problem cuts and most importantly the dynamic constraints, all of which turned out to be necessary to make the approach scale to much larger model sizes.

Since Chapter 6 is based on the Master thesis of Dimitri Bohlender [Boh14] who was supervised by me, I want to say a few words regarding my contribution to the chapter. While overall Dimitri worked independently, major parts of the necessary optimizations are the result of joint discussions and tight interactions. The amendments to the refinement procedure are not part of the thesis and were developed by me. For the implementation of the approach within Storm, I took some inspiration from Dimitri's code but developed the current version independently.

Finally, we turn to [Deh+17]. As this publication presents the work on our probabilistic model checker Storm over several years, it is impossible to sharply measure the contributions of the numerous people involved in the project. I proposed to develop Storm to Joost-Pieter in 2012. Since then, I have been the lead programmer for the project and have supervised more than 15 students during their time as assistants or when writing their Bachelor- or Master-theses revolving around Storm.

Chapter 2

Preliminaries

In this chapter, we establish the formal underpinnings of the remaining chapters. We start by introducing notation and well-known concepts to reason about (probabilistic) systems in a mathematically rigorous way. Next, we formally define the probabilistic models that we address in this thesis and explain their behavior. Then, we lay the foundation for the symbolic representation of these models before we finally discuss algorithmic aids that are used throughout the thesis.

2.1 General Notation

Sets and Functions. If for two sets A and B we have $A \cap B = \emptyset$, we use $A \uplus B$ to express their (disjoint) union. For a set A, we let $\mathscr{P}(A) = \{A' \subseteq A\}$ be the power set of A. We use $|A|$ to refer to the size of a finite set A and, for convenience, we define the size of an infinite set to be ∞.

If A and B are sets, we denote their Cartesian product by $A \times B = \{\langle a, b\rangle \mid a \in A \land b \in B\}$. Even though they are technically not the same, we identify $A \times B \times C$ and both $(A \times B) \times C$ and $A \times (B \times C)$ because there exist canonical isomorphisms between these sets.

A *partial function* $f \colon A \rightharpoonup B$ may leave the value $f(a)$ undefined for arbitrary many elements $a \in A$. If $f(a)$ is indeed undefined, we denote this by $f(a) = \bot$. We denote the usual composition $g(f(x))$ of two functions by $g \circ f$. When a function $f \colon A_1 \times \ldots \times A_n \to B$ depends on several inputs, we use

$$g = f(a_1, \ldots, a_{i-1}, \cdot, a_{i+1}, \ldots, a_n)$$

to denote the function $g\colon A_i \to B$ that fixes all inputs of f to specific values $a_j \in A_j, j \neq i$ except for the remaining argument $a_i \in A_i$. For a function $f\colon A \to B$ and $B' \subseteq B$, we let $f^{-1}(B') = \{a \in A \mid f(a) \in B'\}$ be the *preimage* of B' under f.

We denote the set of real numbers by $\mathbb{R}$, the rationals with $\mathbb{Q}$, and the set of natural numbers (including 0) by $\mathbb{N}$. Given a set of numbers A, we use subscripts to indicate the restriction to the non-negative and positive fragments of A, respectively. For example, $\mathbb{R}_{\geq 0}$ refers to the non-negative reals. We use $[a, b] = \{c \in \mathbb{R} \mid a \leq c \leq b\} \subseteq \mathbb{R}$ for $a, b \in \mathbb{R}$ to denote the interval of reals between a and b including the boundaries. For two sets A and B, we let $A^B = \{f\colon B \to A\}$ denote all functions from B to A. Intuitively, they can be understood as vectors of size $|B|$ with entries in A.

For a set A, we call a function $\mu\colon A \to [0,1]$ with $\sum_{a \in A} \mu(a) = 1$ a *(probability) distribution* on A. The *support* of μ is the set $\mathit{supp}(\mu) = \{a \in A \mid \mu(a) > 0\}$. We let δ_a refer to the *Dirac* distribution on a, i. e. the probability distribution μ with $\mu(a) = 1$ and denote the set of all probability distributions over A by $\mathit{Dist}(A)$.

Relations. A set $\mathcal{R} \subseteq A \times B$ is called a (binary) relation between A and B. If $A = B$, the relation is also simply called a relation on A.

Definition 1 (Properties of Relations)**.** A relation on A is called

» *reflexive* if $\langle a, a\rangle \in \mathcal{R}$ for all $a \in A$,

» *symmetric* if $\langle a, b\rangle \in \mathcal{R} \implies \langle b, a\rangle \in \mathcal{R}$ for all $a, b \in A$,

» *antisymmetric* if $\langle a, b\rangle \in A \wedge \langle b, a\rangle \in \mathcal{R} \implies a = b$ for all $a, b \in A$, and

» *transitive* if $\langle a, b\rangle \in \mathcal{R} \wedge \langle b, c\rangle \in \mathcal{R} \implies \langle a, c\rangle \in \mathcal{R}$ for all $a, b, c \in A$.

As it is one of the key notions of the abstract interpretation framework [CC77], we now define complete lattices.

Definition 2 (Partial Order, Partially Ordered Set, Complete Lattice)**.** A *partial order* $\sqsubseteq$ on a set A is a binary relation on A that is reflexive, antisymmetric and transitive.

A tuple $\langle A, \sqsubseteq\rangle$ is called a *partially ordered set* if $\sqsubseteq$ is a partial order on A.

Let $\langle A, \sqsubseteq \rangle$ be a partially ordered set. An element a is called an *upper bound* (or *lower bound*) for a subset $A' \subseteq A$ if $a' \sqsubseteq a$ (or $a \sqsubseteq a'$) for all $a' \in A'$. An upper (lower) bound a for $A' \subseteq A$ is the *least upper bound* (*greatest lower bound*) if for all upper (lower) bounds a' of A' it is $a \sqsubseteq a'$ ($a' \sqsubseteq a$).

A partially ordered set $\langle A, \sqsubseteq \rangle$ is called a *complete lattice* if there exists a least upper bound and a greatest lower bound for all $A' \subseteq A$.

The importance of complete lattices in abstract interpretation is due the fact that they allow to make use of Kleene's fixed point theorem [Kle52]. Intuitively, this states that the least fixed point $\mathit{lfp}_{\sqsubseteq}(f)$ of a monotonic (w.r.t. $\sqsubseteq$) function $f\colon A \to A$ can be obtained by iterating f on the smallest element of the complete lattice $\langle A, \sqsubseteq \rangle$.

Definition 3 (Equivalence Relation)**.** A binary relation $\mathcal{R}$ on A is called an *equivalence relation* if $\mathcal{R}$ is reflexive, symmetric and transitive. For an equivalence relation $\mathcal{R}$ on A, we write $a_1 \equiv_{\mathcal{R}} a_2$ if $\langle a_1, a_2 \rangle \in \mathcal{R}$ and call a_1 and a_2 $\mathcal{R}$-equivalent.

For an equivalence relation $\mathcal{R}$ on A and a subset $A' \subseteq A$, we let

$$A'/\mathcal{R} = \{\{a_2 \in A' \mid a_1 \equiv_{\mathcal{R}} a_2\} \mid a_1 \in A'\}\,.$$

Then, $A/\mathcal{R}$ is the set of *equivalence classes* of $\mathcal{R}$. We use $[a]_{\mathcal{R}}$ to refer to the (unique) equivalence class of $a \in A$. The *index* of $\mathcal{R}$ is the number of its equivalence classes, i. e. $\mathrm{ind}(\mathcal{R}) = |A/\mathcal{R}|$.

We call $\Pi \subseteq \mathscr{P}(A)$ a *partition of* a set A if

» $\emptyset \notin \Pi$,

» for all $B_1, B_2 \in \Pi$ with $B_1 \neq B_2$ it is $B_1 \cap B_2 = \emptyset$, and

» $\biguplus_{B \in \Pi} B = A$.

Clearly, the equivalence classes of an equivalence relation $\equiv$ on A form a partition. Conversely, a partition of a set A canonically induces an equivalence relation. We therefore identify partitions and equivalence relations and use them interchangeably. For two partitions Π_1 and Π_2 of a set A, we say Π_1 *refines* Π_2, written $\Pi_1 \sqsubseteq \Pi_2$ if $a_1 \equiv_{\Pi_1} a_2$ implies $a_1 \equiv_{\Pi_2} a_2$. Intuitively, this means that Π_1 does not relate elements that were already separated by Π_2. We say that Π_1 *strictly refines* Π_1, written $\Pi_1 \sqsubset \Pi_2$ if $\Pi_1 \sqsubseteq \Pi_2$ and there are two elements a_1, a_2 such that $a_1 \equiv_{\Pi_2} a_2$ but $a_1 \neq\equiv_{\Pi_1} a_2$.

2.2 Probabilistic Models

We now formally define the probabilistic models used in the following chapters. The different model types can be categorized along two dimensions. The first one is whether the formalism has a discrete or continuous notion of time. In the former, the time is treated as abstract and passes in discrete steps whereas in the latter time is modeled as being continuous. The second dimension is the absence or inclusion of nondeterministic choice. Some models allow the environment or a controller to make choices that govern their behavior, while others behave fully probabilistically. Since the remaining models can be understood as restrictions of *Markov reward automata (MRA)*, we introduce them first and then detail how the other models are subsumed.

Definition 4 (Markov Reward Automaton)**.** A *Markov reward automaton (MRA)* is a tuple

$$\mathcal{M} = \langle S, S^0, Act, \Delta, E, r, AP, L \rangle$$

where

» S is a non-empty, countable set of states,

» $S^0 \subseteq S$ is a non-empty set of initial states,

» $\{\tau, \Lambda\} \subseteq Act$ is a countable set of actions,

» $\Delta \subseteq S \times Act \times (S \to \mathbb{R}_{\geq 0}) \times Dist(S)$ is the countable extended transition relation including transition rewards,

» $E\colon S \to \mathbb{R}_{\geq 0}$ is the exit rate function,

» $r\colon S \to \mathbb{R}_{\geq 0}$ is the state reward function,

» AP is a countable set of state labels (also referred to as atomic propositions),

» $L\colon S \to \mathscr{P}(AP)$ is the state labeling function,

such that

(i) for each $s \in S$ there is an element $\langle s, \alpha, \rho, \mu \rangle \in \Delta$,

(ii) for each $\langle s, \alpha, \rho, \mu \rangle \in \Delta$, $\rho(s') > 0 \implies \mu(s') > 0$,

(iii) for each $s \in S$ it is $|\{\langle s, \Lambda, \rho, \mu \rangle \in \Delta\}| \leq 1$,

(iv) $E(s) > 0 \iff \exists \langle s, \Lambda, \rho, \mu \rangle \in \Delta$.

Let $\mathcal{M} = \langle S, S^0, Act, \Delta, E, r, AP, L \rangle$ be an MRA. An element $\langle s, \alpha, \rho, \mu \rangle \in \Delta$ with action α is called a *choice* available (or enabled) in s and $\langle s, s' \rangle$ for $s' \in supp(\mu)$ is called a *transition* (from s to s') Requirement (i) guarantees that every state has at least one choice.

The actions τ and Λ serve special purposes. Choices labeled with τ are called *internal* and model instantaneous, unobservable behavior. In contrast, choices labeled with Λ are called *Markovian* and indicate that time may pass.

Starting from an initial state in S^0, the system moves from state to state using the choices in Δ. When selecting the choice $\langle s, \alpha, \rho, \mu \rangle$, the successor state s' is chosen probabilistically according to the distribution μ and a reward of $\rho(s')$ is gained. Such a choice is called *Markovian* if $\alpha = \Lambda$ and *probabilistic* otherwise. While there may be choices in $\mathcal{M}$ with action $\alpha \notin \{\tau, \Lambda\}$, the traditional interpretation only assigns a semantics to *closed* MRA, which only use the actions τ and Λ. Since other action labels are allowed solely to enable the synchronization of processes during parallel composition [Tim13] with other MRA, the MRA $\mathcal{M}$ can be closed before an actual analysis by *hiding* all other actions in the sense that the corresponding choice are relabeled with τ. In closed MRA, states $s \in S$ are *interactive* if there is an internal choice available in s and *Markovian* if there is a Markovian choice enabled in s. States that are both interactive and Markovian are also referred to as *hybrid* states. As previously mentioned, Markovian choices require time to pass in the source state s. More specifically, the time that is spent in s is (negatively) exponentially distributed with rate $E(s)$. Consequently, the probability to take a Markovian choice at the time the state is entered is 0. This justifies the so-called *maximal progress assumption*: since all internal choices happen instantaneously, the Markovian choices can never be taken in hybrid states and they can therefore be removed (*maximal-progress cut*). The resulting system no longer has hybrid states, but only interactive states that are entered and left instantaneously and Markovian states in which the system spends a random sojourn time. As the reward that is gained while staying in a state s is the product of the sojourn time and the state reward $r(s)$, the system only accumulates (state) rewards in Markovian states.

Condition (ii) states that rewards along choices may only be assigned to existing transitions. Requirement (iii) ensures that there is at most one Markovian choice in each state.

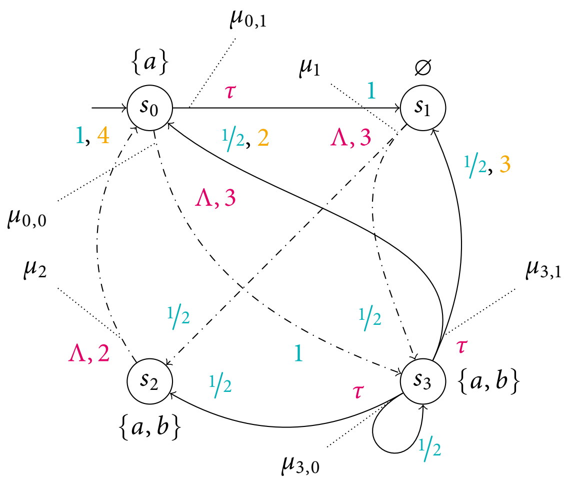


Figure 2.1: An example MRA $\mathcal{M}$.

Since multiple Markovian choices available in one state can always be fused into a single equivalent one, this is no restriction concerning expressivity. Finally, (iv) establishes that non-zero exponential rates only occur in Markovian (or hybrid) states.

We remark that Definition 4 allows loops between interactive states. Such a behavior is called *Zeno* and is typically considered a modeling error, because the system may cycle indefinitely (thereby taking infinitely many steps) without any progression of time. In the course of this thesis, it is irrelevant whether MRA are Zeno or not and we do not consider this problem further.

Example 1. Figure 2.1 shows a example MRA $\mathcal{M} = \langle S, S^0, Act, \Delta, E, r, AP, L \rangle$ with four states $S = \{s_0, s_1, s_2, s_3\}$ and one initial state $S^0 = \{s_0\}$.

To increase readability, we color the different entities as follows:

» actions are magenta,

» probabilities are turquoise, and

» rewards are orange.

Similarly, probabilistic choices are represented by solid arrows and Markovian choices by dashed arrows. We write the exit rates (specified by E) of the states next to the action label Λ of the (unique) Markovian choice (if there is one) and color them just like the action. For instance, it is $E(s_0) = 3$ and $E(s_2) = 2$. We write the transition rewards next to the branching probabilities of the corresponding choices and the state rewards attached to the states. Here, $\mathcal{M}$ does not assign non-zero state rewards and, in general, we omit rewards when they are zero.

Let us look at the two choices $\langle s_3, \tau, \rho_{3,0}, \mu_{3,0} \rangle$ and $\langle s_3, \tau, \rho_{3,1}, \mu_{3,1} \rangle$ available in s_3, where the two probability distributions $\mu_{3,0}$ and $\mu_{3,1}$ are given as

$$\mu_{3,0}(s) = \begin{cases} 1/2 & \text{if } s = s_2 \\ 1/2 & \text{if } s = s_3 \\ 0 & \text{otherwise.} \end{cases} \qquad \mu_{3,1}(s) = \begin{cases} 1/2 & \text{if } s = s_0 \\ 1/2 & \text{if } s = s_1 \\ 0 & \text{otherwise.} \end{cases}$$

Only the second choice assigns non-zero transition rewards via $\rho_{3,1}$:

$$\rho_{3,1}(s) = \begin{cases} 2 & \text{if } s = s_0 \\ 3 & \text{if } s = s_1 \\ 0 & \text{otherwise.} \end{cases}$$

Finally, the states of the $\mathcal{M}$ are labeled with elements from $\mathscr{P}(\{a, b\})$. For instance, $L(s_2) = L(s_3) = \{a, b\}$ and $L(s_1) = \varnothing$.

$\mathcal{M}$'s action set is $Act = \{\tau, \Lambda\}$ and it is therefore trivially closed. However, it has not been subject to the maximal progress cut since in s_0 there is a probabilistic and a Markovian choice. Of the four states, two are Markovian (s_1 and s_2), one is probabilistic (s_3) and one is hybrid (s_0). Applying the maximal-progress cut to $\mathcal{M}$ removes the Markovian choice in s_0 but preserves the nondeterministic choice between the τ choices in s_3.

We call an MRA finite if S, Act, Δ and AP are finite. As a shorthand notation, we use

$$\Delta(s) = \{\langle \alpha, \rho, \mu \rangle \mid \langle s, \alpha, \rho, \mu \rangle \in \Delta\}$$

to refer to the choices enabled in state s and omit ρ if it is the constant zero function.

Furthermore, we let

$$\begin{aligned} pred_{\mathcal{M}}(s) &= \{s' \in S \mid \exists \langle s', \alpha, \rho, \mu \rangle \in \Delta \,.\, s \in supp(\mu)\} \\ succ_{\mathcal{M}}(s) &= \{s' \in S \mid \exists \langle s, \alpha, \rho, \mu \rangle \in \Delta \,.\, s' \in supp(\mu)\} \end{aligned}$$

be the *predecessors* and *successors* of a state $s \in S$, respectively. We write $Act(s) = \{\alpha \in Act \mid \exists \rho, \mu \,.\, \langle s, \alpha, \rho, \mu \rangle\}$ to denote all actions available in a state s and also lift this notion to subsets of states $S' \subseteq S$ by

$$Act(S') = \bigcup_{s \in S'} Act(s).$$

We now introduce additional notation from the literature [Tim13] that makes it easier to describe the step-wise behavior of MRA.

Definition 5 (Extended Action Set, Extended Choices)**.** For an MRA

$$\mathcal{M} = \left\langle S, S^0, Act, \Delta, E, r, AP, L \right\rangle$$

the extended action set Act^{χ} is defined as

$$Act^{\chi} = (Act \setminus \{\Lambda\}) \uplus \{\chi(\lambda) \mid \exists s \in S \wedge E(s) = \lambda \wedge \lambda > 0\}$$

For $\alpha^{\chi} \in Act^{\chi}$, we write $s \xrightarrow{\alpha^{\chi}}_{\rho} \mu$ if

(i) $\alpha^{\chi} \in Act \setminus \{\Lambda\}$ and $\langle s, \alpha^{\chi}, \rho, \mu \rangle \in \Delta$, or

(ii) $\alpha^{\chi} = \chi(\lambda)$ and $\langle s, \Lambda, \rho, \mu \rangle \in \Delta$, $E(s) = \lambda$ and there is *no* $\langle s, \tau, \rho', \mu' \rangle \in \Delta$.

Furthermore, for $\alpha^{\chi} \in Act^{\chi}$ and $\alpha \in Act$ we let

$$\chi^{-1}(\alpha) = \begin{cases} \alpha & \text{if } \alpha \in Act \\ \Lambda & \text{otherwise.} \end{cases}$$

The extended action set introduces a special symbol $\chi(\lambda)$ for all rates $\lambda \in \mathbb{R}_{>0}$ that appear in the MRA. If $\alpha^{\chi} \in Act$, an extended choice $s \xrightarrow{\alpha^{\chi}}_{\rho} \mu$ means that the system can select some probabilistic choice $\langle s, \alpha^{\chi}, \rho, \mu \rangle \in \Delta$ in s that branches according to

μ. However, if $\alpha^\chi = \chi(\lambda)$, it is means that the system may *potentially* spend time in s distributed according to $E(s) = \lambda$ before branching with respect to μ. For the latter, condition (ii) states that there must not be an internal probabilistic choice enabled in s, because if there was such an internal probabilistic choice, even after parallel composition it would still be there and always be taken before time can pass. Therefore, it is never the case that the system may perform the Markovian choice. The condition can hence be thought of as "performing" the maximal progress cut we mentioned earlier. Note that if $\mathcal{M}$ is not closed there still may be other (non-internal) probabilistic choices in s and there still is the possibility that the Markovian choice is removed due to the maximal-progress cut before an analysis. For further details, we refer to [Tim13]. If it is clear from the context that there are no rewards, we omit the subscript from extended choices.

Example 2. Reconsider Example 1. We have

$$\Delta(s_3) = \{\langle\tau, \rho_0, \mu_{3,0}\rangle, \langle\tau, \rho_{3,1}, \mu_{3,1}\rangle\}$$

where ρ_0 denotes the constant zero function on S. Similarly, it is $Act(s_0) = \{\tau, \Lambda\}$, $pred_{\mathcal{M}}(s_0) = \{s_2, s_3\}$ and $succ_{\mathcal{M}}(s_0) = \{s_1, s_3\}$.

There are two distinct exit rates that appear in $\mathcal{M}$, namely $E(s_0) = E(s_1) = 3$ and $E(s_2) = 2$. Hence, the extended action set is $Act^\chi = \{\tau, \chi(2), \chi(3)\}$.

We have $s_3 \xrightarrow{\tau}_{\rho_{3,1}} \mu_{3,1}$ and $s_1 \xrightarrow{\chi(3)}_{\rho_0} \mu_1$. However, we *do **not*** have $s_0 \xrightarrow{\chi(3)}_{\rho_0} \mu_{0,0}$ since there is a choice available in s_0 that is labeled with τ.

The evolution of an MRA is formally captured by the notion of a (timed) path.

Definition 6 (Paths in MRA). Let $\mathcal{M} = \left\langle S, S^0, Act, \Delta, E, r, AP, L\right\rangle$ be an MRA. A *timed path* π starting in a state $s_0 \in S$ is an infinite sequence

$$\pi = s_0 \xrightarrow[\mu_0, \rho_0]{t_0, \alpha_0} s_1 \xrightarrow[\mu_1 \rho_1]{t_1, \alpha_1} s_2 \ldots$$

such that for all $i \in \mathbb{N}$

» $s_i \in S$, $t_i \in \mathbb{R}_{\geq 0}$, $\alpha_i \in Act$

» for

$$\alpha'_i = \begin{cases} \alpha_i & \text{if } \alpha_i \neq \Lambda \\ \chi(E(s_i)) & \text{otherwise} \end{cases}$$

we require $s_i \xrightarrow{\alpha'_i}_{\rho_i} \mu$ and $s_{i+1} \in \mathit{supp}(\mu)$, and

» $t_i > 0$ implies $\alpha_i \in \mathit{Act}^{\chi} \setminus \mathit{Act}$.

An *untimed path* π starting in state $s_0 \in S$ is an infinite sequence

$$\pi = s_0 \xrightarrow[\mu_0,\rho_0]{\alpha_0} s_1 \xrightarrow[\mu_1,\rho_1]{\alpha_1} s_2 \dots$$

if for all $i \in \mathbb{N}$ we have $s_i \xrightarrow{\alpha_i}_{\rho_i} s_{i+1}$.

The sets of timed and untimed paths from s in $\mathcal{M}$ are denoted $\mathit{Paths}^{\Lambda}_{\mathcal{M}}(s)$ and $\mathit{Paths}_{\mathcal{M}}(s)$, respectively, and we omit the subscript if it is clear from the context. We obtain the (timed and untimed) paths $\mathit{Paths}_{\mathcal{M}}$ and $\mathit{Paths}^{\Lambda}_{\mathcal{M}}$ of $\mathcal{M}$ by joining the paths of all initial states. The non-empty finite prefixes $\widehat{\pi}$ of (timed and untimed) paths are also referred to as finite (timed or untimed) paths and are denoted $\mathit{FPaths}^{\Lambda}_{\mathcal{M}}$ and $\mathit{FPaths}_{\mathcal{M}}$, respectively. In this case, the refer to the last state s of $\widehat{\pi}$ by $\mathit{last}(\widehat{\pi})$.

Example 3. Reconsider the MRA $\mathcal{M}$ from Example 1. A timed path $\pi \in \mathit{Paths}^{\Lambda}_{\mathcal{M}} = \mathit{Paths}^{\Lambda}_{\mathcal{M}}(s_0)$ is, for instance,

$$\pi = s_0 \xrightarrow[\mu_{0,1},\rho_0]{0,\tau} s_1 \xrightarrow[\mu_1,\rho_0]{\frac{3}{2},\Lambda} s_2 \xrightarrow[\mu_2,\rho_2]{\frac{1}{4},\Lambda} s_0 \dots$$

where

$$\rho_2(s) = \begin{cases} 4 & \text{if } s = s_0 \\ 0 & \text{otherwise.} \end{cases}$$

We obtain the untimed path π' from π by abstracting from the timing information:

$$\pi' = s_0 \xrightarrow[\mu_{0,1},\rho_0]{\tau} s_1 \xrightarrow[\mu_1,\rho_0]{\Lambda} s_2 \xrightarrow[\mu_2,\rho_2]{\Lambda} s_0 \dots$$

As a next step, we introduce Markov automata (MA) as MRA without rewards.

Definition 7 (Markov Automaton)**.** An MRA

$$\mathcal{M} = \langle S, S^0, Act, \Delta, E, r, AP, L \rangle$$

is called a Markov automaton (MA), if

» $r(s) = 0$ for all states $s \in S$,

» for every $\langle s, \alpha, \rho, \mu \rangle \in \Delta$, ρ is the constant 0 function.

We write it as a tuple

$$\mathcal{M} = \langle S, S^0, Act, \Delta, E, AP, L \rangle$$

where we omit the state reward function r and let the transition relation $\Delta \subseteq S \times Act \times Dist(S)$ not contain transition rewards.

We continue to use the notion of extended choices for MA (and all models they subsume) but omit the transition reward subscript.

Probabilistic automata (PA) are MA without Markovian choices. They contain nondeterminism, but no time is spent in any state and the notion of extended choices becomes superfluous as there are not Markovian choices.

Definition 8 (Probabilistic Automaton)**.** An MA

$$\mathcal{M} = \langle S, S^0, Act, \Delta, E, AP, L \rangle$$

is a probabilistic automaton (PA), if there is no $\langle s, \Lambda, \mu \rangle \in \Delta$. We write it as a tuple

$$\mathcal{M} = \langle S, S^0, Act, \Delta, AP, L \rangle$$

without the exit rate function.

We remark that PA are similar to Markov decision processes (MDPs) [Bel57] but allow several equally labeled choices per state.

In sharp contrast to PA, continuous-time Markov chains (CTMCs) have *only* Markovian choices and have exactly one initial state.

Definition 9 (Continuous-Time Markov Chain)**.** An MA

$$\mathcal{M} = \left\langle S, S^0, Act, \Delta, E, AP, L \right\rangle$$

is a CTMC, if

(i) $S^0 = \left\{s^0\right\}$, and

(ii) there is no $\langle s, \alpha, \mu \rangle \in \Delta$ with $\alpha \neq \Lambda$.

We write it as a tuple

$$\mathcal{C} = \left\langle S, s^0, \mathbf{P}, E, AP, L \right\rangle$$

with $\mathbf{P}\colon S \to Dist(S)$ such that $\mathbf{P}(s)$ is the uniquely determined distribution of s.

As the definition of MRA (Definition 4) forbids deadlock states and allows at most one Markovian choice per state, every state in a CTMC has *exactly* one Markovian choice. Therefore, we ease the notation by moving from the transition relation Δ to a transition *function* $\mathbf{P}$ that maps states to their unique probability distributions. Note that the sojourn times of states are still distributed according to the exit rate function E.

Finally, we define discrete-time Markov chains (DTMCs) as PA in which every state has exactly one choice that is required to be probabilistic.

Definition 10 (Discrete-Time Markov Chain)**.** A PA

$$\mathcal{M} = \left\langle S, S^0, Act, \Delta, AP, L \right\rangle$$

is a discrete-time Markov chain, if

(i) $S^0 = \left\{s^0\right\}$, and

(ii) $Act = \{\tau, \Lambda\}$,

	deterministic	nondeterministic
discrete time	DTMCs	PA
continuous time	CTMCs	MA

Table 2.1: Overview of model types.

(iii) there is no $\langle s, \Lambda, \mu \rangle \in \Delta$, and

(iv) for every $s \in S$ there is exactly one $\langle s, \tau, \mu \rangle \in \Delta$.

We write it as a tuple

$$\mathcal{D} = \left\langle S, s^0, \mathbf{P}, AP, L \right\rangle$$

with $\mathbf{P}\colon S \to \mathit{Dist}(S)$ such that $\mathbf{P}(s)$ is the uniquely determined distribution of s.

Just as for CTMCs, we use **P** to map states to their unique distributions.

We conclude this section with Table 2.1, which gives an overview of the model types (without rewards) that we just presented categorized by which notion of time they have and whether or not they include nondeterministic choices.

2.3 Measures and Logics

In order to formalize the behaviour of probabilistic models, two key notions are probability spaces and probability measures.

Definition 11 (Probability Space). A *probability space* is a tuple $\langle \Omega, \mathcal{F}, P \rangle$ consisting of

» a non-empty *sample space* Ω,

» a set of *events* $\mathcal{F} \subseteq \mathscr{P}(\Omega)$ with

(i) $\Omega \in \mathcal{F}$,

(ii) $E \in \mathcal{F} \implies \Omega \setminus E \in \mathcal{F}$, and

(iii) if $E_i \in \mathcal{F}$ for $i \in \mathbb{N}$, then $\bigcup_{i \in \mathbb{N}} E_i \in \mathcal{F}$.

» $P: \mathcal{F} \to [0,1]$ is a *probability measure* with

(i) $P(\Omega) = 1$, and

(ii) $P(\bigcup_{i \geq 0} E_i) = \sum_{i \geq 0} P(E_i)$ if $E_j \cap E_k = \varnothing$ for all $i \neq k$.

Intuitively, a probability space models a random experiment. Such an experiment is subject to uncertainty and its outcome is therefore not uniquely determined. Instead, there is a set of possible outcomes Ω that is also called the sample space. Instead of letting the probability measure P assign probabilities to outcomes directly, it maps events, i. e. sets of outcomes, to probabilities. This is done, because it may be the case that for continuous sample spaces the probability for each outcome is zero, whereas the probability of a *set* of outcomes may be non-zero.

The first requirement on the probability measure P expresses that the probability that the experiment yields any of the possible outcomes is one. The second one is typically referred to as countable additivity and states that the probability of an event that can be decomposed into countably many non-overlapping events E_i is the sum of probabilities of the latter.

The first and third restriction on the set of events are both needed for well-definedness of the probability measure: in order to assign probabilities to sets of outcomes, they have to be included in the event set. Such sets are called *measurable*. Finally, (ii) ensures that the probability of any event and its complement always add to one. A set satisfying the constraints for the event set are also referred to as *σ-algebras*. Together with a sample space Ω, a σ-algebra $\mathcal{F}$ forms a measurable space $\langle \Omega, \mathcal{F} \rangle$. It is well-known that $\langle \mathbb{R}, \mathscr{P}(\mathbb{R}) \rangle$ is *not* a measurable space as the power set of the reals contains the so-called Vitali sets that cannot be assigned any meaningful probability. However, $\langle \mathbb{R}, \mathfrak{B}_{\mathbb{R}} \rangle$ is in fact a measurable space, where $\mathfrak{B}_{\mathbb{R}}$ is the *Borel σ-algebra* over $\mathbb{R}$, the smallest σ-algebra over the reals that subsumes all intervals.

For the probabilistic models defined above, it is possible to construct unique probability measures. Here, we refrain from restating the detailed theory as it is not relevant in the course of the thesis. Instead, we refer to the literature [BK08; Qua16] for details on the precise construction and rather describe the intuition of the probability measures and involved probability spaces.

Let us start with with the probability measure for a DTMC $\mathcal{D} = \left\langle S, s^0, \mathbf{P}, AP, L \right\rangle$. As for discrete-time models the time is irrelevant, the sample space of the measurable space are the untimed paths $Paths_{\mathcal{D}}$ in $\mathcal{D}$. The events are then constructed via a cylinder set construction [BK08]. Intuitively, the cylinder set of a finite path $\widehat{\pi}$ is the set of all infinite paths that start with $\widehat{\pi}$ and then proceed arbitrarily. The corresponding σ-algebra is the smallest one that contains the cylinder sets of all finite paths in $\mathcal{D}$. Fixing the probabilities of the cylinder sets to the product of the transition probabilities along the corresponding finite path, there exists a unique extension to a probability measure $\Pr_{\mathcal{D}}$ that assigns probabilities to (measurable) sets of (untimed) paths. For a CTMC $\mathcal{C}$, the construction for the probability measure $\Pr_{\mathcal{C}}$ is slightly more involved as it needs to consider timed paths to enable reasoning over its timed behavior.

For the models that incorporate nondeterministic choice the probabilities of events depend on the resolution of nondeterminism. Formally, this happens via *schedulers*.

Definition 12 (Scheduler). Let $\mathcal{M} = \left\langle S, S^0, Act, \Delta, E, r, AP, L \right\rangle$ be an MRA. A *scheduler* σ for $\mathcal{M}$ is a function

$$\sigma \colon FPaths^{\Lambda}_{\mathcal{M}} \to Dist(Act \times (S \to \mathbb{R}_{\geq 0}) \times Dist(S))$$

that maps finite paths $\widehat{\pi}$ to a distribution μ such that $supp(\mu) \subseteq \Delta(last(\widehat{\pi}))$. The set of all schedulers for $\mathcal{M}$ is denoted by $\mathfrak{S}_{\mathcal{M}}$.

Schedulers for PA can be defined analoguously on untimed paths. By mapping finite paths to a distribution over the available choices, schedulers resolve the nondeterminism completely and the "application" of a scheduler σ to an MRA $\mathcal{M}$ results in a fully probabilistic (timed) system $\mathcal{M}^{\sigma}$. For these systems, the probability measure constructions above can be employed. As the probability measure for an MRA or PA $\mathcal{M}$ depends on a scheduler σ for $\mathcal{M}$ we denote it by $\Pr^{\sigma}_{\mathcal{M}}$. Typically, for nondeterministic models one is interested in the extremal probabilities. We therefore define the *minimal* and *maximal* probability of a (measurable) set of paths Π, respectively, as

$$\Pr^{-}_{\mathcal{M}}(\Pi) = \inf_{\sigma \in \mathfrak{S}_{\mathcal{M}}} \Pr^{\sigma}_{\mathcal{M}}(\Pi) \text{ and } \Pr^{+}_{\mathcal{M}}(\Pi) = \sup_{\sigma \in \mathfrak{S}_{\mathcal{M}}} \Pr^{\sigma}_{\mathcal{M}}(\Pi).$$

To unambiguously formulate properties over formal models, typically temporal logics are employed. In the non-probabilistic setting, the most famous logics are the linear temporal logic (LTL) [Pnu77] and the computation tree logic (CTL) [CE81]. While the former uses a linear view of time, the latter is interpreted over computation trees and therefore employs a branching-time view. One way to state properties of probabilistic systems is to give regular LTL a probabilistic interpretation (typically referred to as probabilistic linear temporal logic (PLTL)). That is, instead of a binary answer that a system satisfies or violates a given LTL formula φ, we use the probability measures over paths to determine the probability mass of the set of paths characterized by φ. Another route is to extend CTL to account for the probabilistic and timed behavior. This leads to the probabilistic computation tree logic (PCTL) [HJ94] that essentially replaces the usual CTL path quantifiers by a probability operator $\mathsf{P}_{\bowtie\lambda}(\varphi)$ where φ is a PCTL path formula. Instead of stating that the nested path formula φ holds for at least one or all paths, it states that the probability to satisfy the path formula must meet the bound $\bowtie\lambda$. PCTL is also sometimes defined to include the bounded-until operator $\mathsf{U}^{\leq k}$ that expresses that certain states are reached within k (discrete) steps. The logic CSL adapts this for timed models such as CTMCs and MA by considering the sojourn time in states rather than the number of steps.

The most fundamental property that is expressible in all of the above logics is *reachability*. Ultimately, many of the other verification tasks can be reduced to reachability. Since it is also the single most important property in the course of this thesis, we formally define it as follows. For a given set $T \subseteq S$, we let

$$\lozenge T(s,\mathcal{M}) = \left\{ \pi = s_0 \xrightarrow[\mu_0,\rho_0]{\alpha_0} s_1 \xrightarrow[\mu_1,\rho_1]{\alpha_1} s_2 \ldots \in \mathit{Paths}_{\mathcal{M}}(s) \mid \exists i \in \mathbb{N} \,.\, s_i \in T \right\}$$

be all (untimed) paths that start in s_0 and eventually reach some state in T in $\mathcal{M}$. Here, the set T may be given either explicitly or implicitly, for instance by means of a formula that evaluates to a truth value in each state. We omit the state s and the model $\mathcal{M}$ if they are clear from the context and write $s \vDash_{\mathcal{M}} \exists\lozenge T$ if $\lozenge T(s,\mathcal{M}) \neq \emptyset$.

The aforementioned logics can be extended to allow for reasoning about rewards in the system. Typically, these extensions (at least) involve the expected reward that is

» accumulated until some set of target states is reached,

» accumulated within the first k steps (or time units), or

» obtained at exactly step (or time point) k.

2.4 Variables and Expressions

Large parts of this thesis are concerned with handling probabilistic models *symbolically*. In this context, the model is often encoded in terms of variables and expressions similar to programming languages. For our presentation, we assume each variable $x \in \mathit{Var}$ in a set of variables is typed and has (potentially infinite) domain $\mathit{Dom}(x)$.

Definition 13 (Variable Valuation)**.** A *variable valuation* for a set of variables Var is a function

$$v \colon \mathit{Var} \to \mathit{Dom}(\mathit{Var})$$

where

$$\mathit{Dom}(\mathit{Var}) = \bigcup_{x \in \mathit{Var}} \mathit{Dom}(x)$$

such that $v(x) \in \mathit{Dom}(x)$ for all $x \in \mathit{Var}$. We use $\mathit{Val}(\mathit{Var})$ to denote the set of all variable valuations for the variables Var and $\eta@v$ to denote the value of the expression η over the variables Var when all variables are replaced by their value according to v. Furthermore, if $v_i \in \mathit{Val}(\mathit{Var}_i)$ for $i \in \{1,2\}$ and for all $x \in \mathit{Var}_1 \cap \mathit{Var}_2$ it is $v_1(x) = v_2(x)$, then

$$(v_1 \oplus v_2) \in \mathit{Val}(\mathit{Var}_1 \cup \mathit{Var}_2)$$
$$(v_1 \oplus v_2)(x) = \begin{cases} v_1(x) & \text{if } x \in \mathit{Var}_1 \\ v_2(x) & \text{if } x \in \mathit{Var}_2 \end{cases}$$

is the *joint variable valuation*.

We refrain from precisely defining expressions and treat them abstractly if possible. We simply assume that all expressions are well-typed and evaluate to a value of the same type when all contained variables are substituted by a value, for example through the use of a variable valuation v. We let $\mathit{Bxp}(\mathit{Var})$, $\mathit{Qxp}(\mathit{Var})$ and $\mathit{Exp}(\mathit{Var})$ refer to the set of expressions that evaluate to a *Boolean*, some number or an arbitrary value, respectively. For a Boolean expression $b \in \mathit{Bxp}(\mathit{Var})$ and a variable valuation $v \in \mathit{Val}(\mathit{Var})$, we write $v \models b$ if $b@v = \mathit{true}$.

2.5 Binary Decision Diagrams

In this section we show how to represent probabilistic systems using binary decision diagrams, a data structure that is able to represent structured information efficiently. For further details that are beyond the scope of this thesis, we refer to [Par03].

Let $Var = \{x_1, \ldots, x_n\}$ be a set of Boolean variables and

$$x_1 \prec x_2 \prec \ldots \prec x_n$$

be a total ordering of the variables in Var.

Definition 14 (Multi-Terminal Binary Decision Diagram)**.** A multi-terminal binary decision diagram (MTBDD) M over $\langle Var, \prec \rangle$ is a tuple

$$\mathsf{M} = \langle V = V_I \uplus V_T, then, else, var, val, n_0 \rangle$$

where

- » V is a finite set of *nodes*, partitioned into into the *inner nodes* V_I and the terminal nodes V_T,
- » two successor functions $then, else\colon V_I \to V$ that assign to each inner node $n \in V$ a *then*-successor $then(n) \in V$ and an *else*-successor $else(n) \in V$,
- » a variable labeling function $var\colon V_I \to Var$ that assigns to each inner node $n \in V$ a variable $var(n) \in Var$,
- » a value function $val\colon V_T \to \mathbb{R}$ that assigns to each terminal node $n \in V$ a real value $val(n)$, and
- » a root node $n_0 \in V$.

Furthermore, we require that the labeling is consistent in the sense that for all $n \in V_I$

$$var(n) \prec var(then(n)) \qquad \text{and} \qquad var(n) \prec var(else(n))$$

where for readability we extend var to terminal nodes by $var(n_T) = \perp$ and let $x \prec \perp$ for every $x \in Var$.

A BDD is an MTBDD for which $val(n) \in \{0,1\}$ for all $n \in V_T$. We use $Var(\mathsf{M}) = \bigcup_{n \in V_I} \{var(n)\}$ to denote the variables that appear in M and $\mathrm{root}(\mathsf{M})$ to refer to the root node n_0 of M.

Note that every node of an MTBDD is itself the root of an MTBDD. We therefore often identify an MTBDD and its root node.

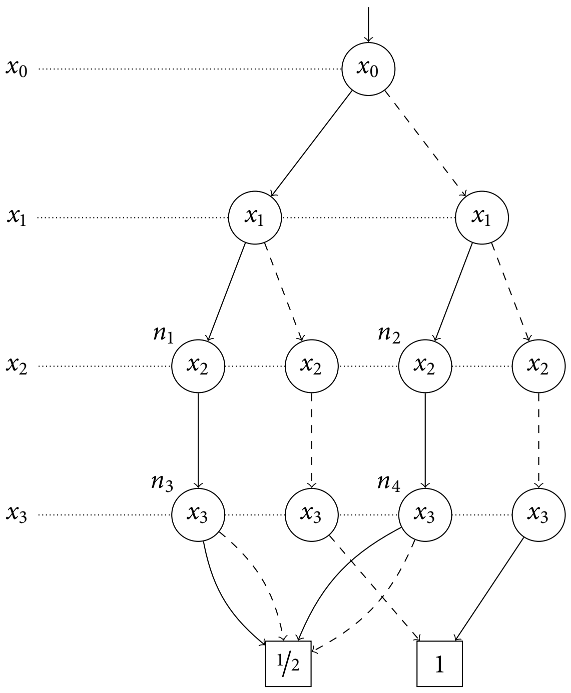


Figure 2.2: An example MTBDD M.

Example 4. Figure 2.2 shows an example MTBDD

$$\mathsf{M} = \langle V = V_I \uplus V_T, \mathit{then}, \mathit{else}, \mathit{var}, \mathit{val}, n_0 \rangle$$

over the four variables $\mathit{Var} = \{x_0, x_1, x_2, x_3\}$ using the ordering

$$x_0 \prec x_1 \prec x_2 \prec x_3.$$

The inner nodes are drawn as circles and the terminal nodes as rectangles containing their value assigned by *val*. We draw equally labeled nodes in a layered fashion and give the labels of each level of variables. The functions *then* and *else* are represented visually: for each node n, the solid edge points to the *then*-successor $\mathit{then}(n)$ and the dashed edge to the *else*-successor $else(n)$. In general, unless stated otherwise we omit edges to the constant zero terminal node for readability.

Let us fix an MTBDD $\mathsf{M} = \langle V = V_I \uplus V_T, then, else, var, val, n_0 \rangle$ for the rest of the chapter. The semantics of M is a function $f_{\mathsf{M}} \colon Val(Var) \to \mathbb{R}$. The function value of f_{M} for an input valuation $v \in Val(Var)$ can be obtained as follows. We start at the root $n_0 = \mathrm{root}(\mathsf{M})$. If this node is a terminal node, the function value is given by $val(n_0)$. If not, we move to the *then*-successor if $v(var(n_0)) = 1$ and to the *else*-successor otherwise. Continuing this process from the new node ultimately yields a terminal node $n \in V_T$ that specifies the function value under v as $val(n)$. If $Var(\mathsf{M})$ is a proper subset of Var, we may omit the missing variables from the signature of the associated function f_{M}.

Example 5. Reconsider the MTBDD from Example 4. It represents the function

$$f_{\mathsf{M}}(x_0, x_1, x_2, x_3) = \begin{cases} 1/2 & \text{if } (x_0 \wedge x_1 \wedge x_2) \vee (\neg x_0 \wedge x_1 \wedge x_2) \\ 1 & \text{if } (x_0 \wedge \neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (\neg x_0 \wedge \neg x_1 \wedge \neg x_2 \wedge x_3) . \end{cases}$$

MTBDDs are used to efficiently represent highly structured models. To this end, implementations use *reduced* versions only. Through two simple reduction rules [Jr78], M does not possess two distinct nodes $n_1, n_2 \in V$ such that $f_{n_1} = f_{n_2}$. In other words, there are no redundant nodes in the sense that they represent the same subfunctions. Reducing MTBDDs in this way has another fundamental advantage. It can be shown that two reduced MTBDDs represent the same functions if and only if they are equal up to isomorphism. In the further course, we consider MTBDDs to always be reduced, which is guaranteed by most implementations such as CUDD [Som] and SYLVAN [Dij16]. We often define and identify an MTBDD M and the function f_{M} it represents if there is no ambiguity, which is possible since (in the reduced form) the MTBDD M for f_{M} is unique. Sometimes, we only specify the function values for parts of the domain and implicitly map the remaining variable valuations to 0.

Example 6. Reconsider the MTBDD from Example 4. It contains nodes that are superfluous. For instance, both successors of nodes n_3 and n_4 are identical and the nodes may therefore be "skipped". Similarly, the nodes n_1 and n_2 are roots of isomorphic subgraphs and therefore represent the same subfunctions.

Figure 2.3 shows the reduced MTBDD M' that represents the same function as M (see Example 5). In particular, we see that there is no node whose two successors are identical and that the node n is shared to eliminate isomorphic subgraphs.

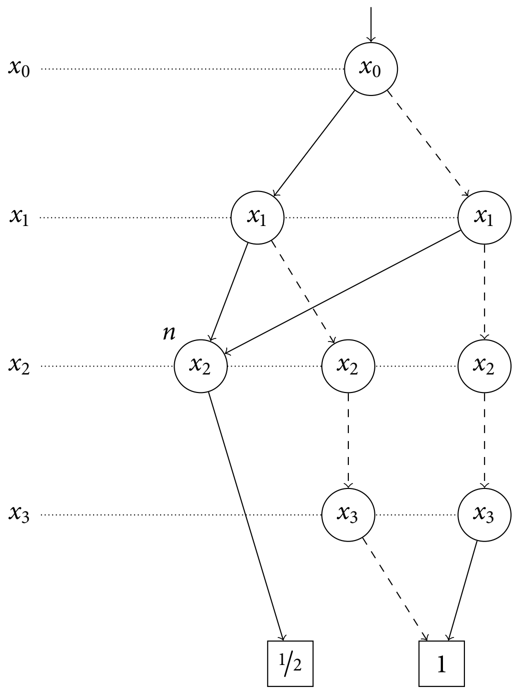


Figure 2.3: A reduced MTBDD M'.

For ease of presentation, we also use

$$\mathsf{M}(v_1,\dots,v_k) = \mathsf{M}(\bigoplus_{i=1}^{k} v_i) = f_{\mathsf{M}}(\bigoplus_{i=1}^{k} v_i) \text{ where } v_i \in \mathit{Val}(\mathit{Var}_i) \text{ with } \biguplus_{i=1}^{k} \mathit{Val}(\mathit{Var}_i) = \mathit{Var}.$$

to refer to the function value of f_{M} at the point given by the variable valuations $v_1,\dots,v_k$.

Operations. For $\mathit{Var}' \subseteq \mathit{Var}$ and $v \in \mathit{Val}(\mathit{Var}')$, we define the *generalized cofactor* $\mathsf{M}|_v = \mathsf{M}'$ of M with respect to v as the MTBDD M' representing

$$f_{\mathsf{M}'}\colon \mathit{Val}(\mathit{Var} \setminus \mathit{Var}') \to \mathbb{R}$$
$$f_{\mathsf{M}'}(v') = f_{\mathsf{M}}(v' \oplus v).$$

In other words, the generalized cofactor amounts to the partial function application in which all variables $x \in \mathit{Var}'$ are fixed to $v(x)$. If the variables in Var' are the smallest in Var with respect to the total ordering $<$, the cofactor operation amounts to descending from $\text{root}(\mathsf{M})$ according to v and returning the resulting node.

In the scope of this thesis, we make use of several operations on MTBDDs. The following three operations "create" decision diagrams (DDs) from other entities:

» $\textsc{Const}(c)$ for $c \in \mathbb{R}$ is the MTBDD that represents the constant c function.

» for a variable valuation $v \in \mathit{Val}(\mathit{Var})$, we use $\textsc{Encode}(v)$ to denote the BDD B such that $\mathsf{B}(v') = 1 \iff v = v'$.

» for a variable x, $\textsc{Id}(x)$ is the BDD B with $\mathsf{B}(v) = 1 \iff v(x) = 1$.

Let $\mathsf{M}, \mathsf{M}_1, \mathsf{M}_2$ be MTBDDs over $\langle \mathit{Var}, < \rangle$.

» if M is a BDD, $\textsc{Not}(\mathsf{M})$ is the BDD B such that $\mathsf{B}(v) = 1 \iff \mathsf{M}(v) = 0$,

» $\textsc{Apply}(\bullet, \mathsf{M}_1, \mathsf{M}_2)$ with $\bullet\colon \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ yields the MTBDD representing the function $f_{\mathsf{M}_1} \bullet f_{\mathsf{M}_2}$. The operation $\bullet$ may, for example, be addition or multiplication, but also min, max and comparison operators such as $\leq$ or $=$. For comparison operations, the result is a BDD. For the division operation, we define $\frac{0}{0} = 0$. If M_1 and M_2 are in fact BDDs, then $\bullet$ may also have the type $\bullet\colon \mathbb{B} \times \mathbb{B} \to \mathbb{B}$, which allows for the usual logical connectives.

» $\textsc{Abstract}(\bullet, \mathit{Var}', \mathsf{M})$ with a commutative and associative operation $\bullet\colon \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ and a set of variables $\mathit{Var}' \subseteq \mathit{Var}$ yields the MTBDD M' representing the function

$$f_{\mathsf{M}'} = \bigodot_{v \in \mathit{Val}(\mathit{Var}')} f_{\mathsf{M}|_v}$$

Similar to $\textsc{Apply}$, we allow operations $\bullet\colon \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ if M is a BDD. The most important instantiations of this operation are the ones with $\circ \in \{\vee, +\}$, that are denoted with $\textsc{ExistsAbstract}(\mathit{Var}', \mathsf{M})$ and $\textsc{SumAbstract}(\mathit{Var}', \mathsf{M})$, respectively.

» $\textsc{Ite}(\mathsf{B}, \mathsf{M}_1, \mathsf{M}_2)$ is the if-then-else operation that yields the M'

$$\mathsf{M}' = \textsc{Apply}\left(+, \textsc{Apply}\left(\times, \mathsf{B}, \mathsf{M}_1\right), \textsc{Apply}\left(\times, \textsc{Not}\left(\mathsf{B}\right), \mathsf{M}_2\right)\right)$$

that yields the function values of M_1 for the variable valuations for which B evaluates to true and the values of M_2 otherwise. As a special case, we allow a variable x instead of the BDD B as the first argument to abbreviate $\textsc{Ite}(\textsc{Id}(x), \mathsf{M}_1, \mathsf{M}_2)$. Note that if $x \prec \mathit{Var}(\mathsf{M}_1), \mathit{Var}(\mathsf{M}_2)$, this operation amounts to creating a new node labeled with x and whose *then* and *else* successors are nodes $\mathrm{root}(\mathsf{M}_1)$ and $\mathrm{root}(\mathsf{M}_2)$, respectively.

» $\textsc{Permute}(\mathsf{M}, \sigma)$ with a permutation σ of $\{1, \ldots, n\}$ is the MTBDD M' representing the function

$$f_{\mathsf{M}'}(v) = f_{\mathsf{M}}(\sigma^{-1}(v)) \text{ where } \sigma^{-1}(v)(x) = v(\sigma^{-1}(x))$$

As a special case that appears frequently, we let $\textsc{Rename}(\mathsf{M}, \mathit{Var}_1, \mathit{Var}_2)$ for variable sets $\mathit{Var}_1, \mathit{Var}_2 \subseteq \mathit{Var}$ with

- $\mathit{Var}_1 = \{y_1, \ldots, y_k\}, y_1 \prec \ldots \prec y_k$,
- $\mathit{Var}_2 = \{y'_1, \ldots, y'_k\}, y'_1 \prec \ldots \prec y'_k$,
- $(\mathit{Var}(\mathsf{M}) \setminus \mathit{Var}_1) \cap \mathit{Var}_2 = \varnothing$

be defined as $\textsc{Permute}(\mathsf{M}, \sigma')$ where

$$\sigma'(i) = \begin{cases} j & \text{if } \exists \ell \,.\, y_\ell = x_i \wedge y'_\ell = x_j \\ i & \text{otherwise} \end{cases}$$

» $\textsc{TopVar}(\mathsf{M})$ is the variable $x = \mathit{var}(n_0)$ where n_0 is the root node of M. We also use $\textsc{TopVar}(\mathsf{M}_1, \ldots, \mathsf{M}_n) = \min \{x \in \mathit{Var} \mid \exists i \,.\, x = \textsc{TopVar}(\mathsf{M}_i)\}$ to retrieve the smallest variable (with respect to the total variable ordering) among the top variables of all M_i.

Based on the fact that $f_{\mathsf{M}_1 \circ \mathsf{M}_2}$ is actually represented by $\mathsf{M}_1 \circ \mathsf{M}_2$, we use the infix notation for the arithmetic and logical operations. For example, we write $\mathsf{M}_1 + \mathsf{M}_2$ instead of $\textsc{Apply}(+, \mathsf{M}_1, \mathsf{M}_2)$ and $\neg\mathsf{M}$ instead of $\textsc{Not}(\mathsf{M})$.

Implementation Details. We want to mention a few implementation details related to MTBDDs before discussing how to encode probabilistic models. Ultimately, it is these details that allow operations on MTBDDs to be efficient. Consequently, they are employed in slight variations across all libraries providing functionality revolving around MTBDDs.

As previously mentioned, MTBDDs are stored in a canonical reduced form that avoids nodes that represent the same (sub)function. To establish this in actual implementations, a so-called *unique table* is used. Essentially, it is a (hash) map that maps a triple $\langle x, n', n'' \rangle$ to a node n if $\mathit{then}(n) = n'$, $\mathit{else}(n) = n''$, $\mathit{var}(n) = x$ and the node n has previously been encountered. Before creating a new node for the corresponding function, the unique table is checked for an already existing node. Only if there is none, a new node for this function is created and otherwise the previous node is reused. Keeping nodes unique this way not only allows for a canonic representation but also reduces function equality checks to a simple check whether two nodes are the very same.

Unlike the unique table, which is necessary to guarantee reducedness, the second (hash) map that is maintained is not critical to the operation but can be seen as an optimization. The *compute table* constitutes a cache that stores intermediate results for previous computations. More specifically, it maps entries $\langle \bullet, n', n'' \rangle$ to a node n if $f_n = f_{n'} \bullet f_{n''}$. During the recursive descent necessary to process most DD-related operations (like APPLY), results of previous operations can be potentially reused by checking the compute table for corresponding entries.

Representing Markov Chains. Before we start describing the encoding of probabilistic systems in terms of DDs, we want to stress that we assume the system to be given explicitly. However, in practice, the system is typically specified in terms of a high-level modeling language (like **PRISM** or the language **JANI** that is the topic of Chapter 3). The details of this encoding are not important for this thesis and we refer to [Par03].

We now discuss how to represent probabilistic models without nondeterminism as a precursor to models with nondeterminism. For this, let $\mathcal{D} = \langle S, s^0, \mathbf{P}, AP, L \rangle$ be a DTMC. To encode the model, we use the set of Boolean variables

$$\mathit{Var}_{\mathcal{D}} = \underbrace{\{\mathfrak{s}_1, \ldots, \mathfrak{s}_n\}}_{\mathcal{S}} \uplus \underbrace{\{\mathfrak{s}'_1, \ldots, \mathfrak{s}'_n\}}_{\mathcal{S}'} \qquad \text{where} \qquad n \geq \lceil \log_2 |S| \rceil.$$

In practice, model checkers typically choose the variable ordering

$$x_1 < x'_1 < x_2 < x'_2 < \ldots < x_n < x'_n$$

which empirically was shown to produce small decision diagrams for many models and enable efficient variable renaming, but this ordering is merely a heuristic and the optimal variable order differs from model to model. For the most part of our presentation, the variable order is arbitrary and we will explicitly state when it becomes relevant. We encode (bit) vectors as DDs over the unprimed variables $\mathcal{S}$. Throughout this section, we assume an encoding $\langle\cdot\rangle$ of entities in terms of variable valuations. For example, $\mathcal{S} \leftarrow \langle s \rangle$ denotes the encoding of state $s \in S$ in terms of a unique valuation of the unprimed variables $\mathcal{S}$. As we choose the sizes of the variable set $\mathcal{S}$ appropriately, such a mapping always exists. This may, for example, be a binary encoding of an arbitrary, but fixed indexing of the states. We can then encode

» S as a BDD B_S such that

$$\mathsf{B}_S(\mathcal{S} \leftarrow \langle s \rangle) = 1 \iff s \in S,$$

» s^0 as a BDD $\mathsf{B}_{s^0} = \textsc{Encode}\left(\mathcal{S} \leftarrow \left\langle s^0 \right\rangle\right)$ with

$$\mathsf{B}_{s^0}(\mathcal{S} \leftarrow \langle s \rangle) = 1 \iff s = s^0,$$

» L as $|AP|$ BDDs where for each atomic proposition $a \in AP$ we have

$$\mathsf{B}_a(\mathcal{S} \leftarrow \langle s \rangle) = 1 \iff a \in L(s)$$

where all DDs are over the variables $\mathcal{S}$. It remains to encode the transition probability function $\mathbf{P}$ of $\mathcal{D}$. Here, we make use of both the unprimed and the primed variables with the intuition that the unprimed variables encode the source states and the primed variables encode the target states. $\mathcal{S}$ can also be thought of as *row variables* and $\mathcal{S}'$ as *column variables* when viewing $\mathbf{P}$ as a matrix. Formally, we use an MTBDD $\mathsf{M}_\mathbf{P}$ over *Var* that represents the function (or matrix)

$$\mathsf{M}_\mathbf{P}(\mathcal{S} \leftarrow \langle s \rangle, \mathcal{S}' \leftarrow \langle s' \rangle) = \mathbf{P}(s, s')$$

Representing a CTMC is very similar. The only thing that has to be stored additionally is the MTBDD-representation M_E of the exit rate function:

$$\mathsf{M}_E(\mathcal{S} \leftarrow \langle s \rangle) = E(s).$$

Example 7. Figure 2.4 shows a DTMC $\mathcal{D}$ that is structurally very similar to the MRA $\mathcal{M}$. Since it has four states, we need 2 Boolean variables to encode the individual states. Let us use the variable sets $\mathcal{S} = \{\mathfrak{s}_1, \mathfrak{s}_0\}$ and $\mathcal{S}' = \{\mathfrak{s}_1', \mathfrak{s}_0'\}$. For readability, we choose the variable ordering $\mathfrak{s}_1 < \mathfrak{s}_0 < \mathfrak{s}_1' < \mathfrak{s}_0'$ that encodes the source states of the transitions on top of the transition MTBDD and the target states at the bottom. Furthermore, we assume the usual binary encoding of the state indices to map states

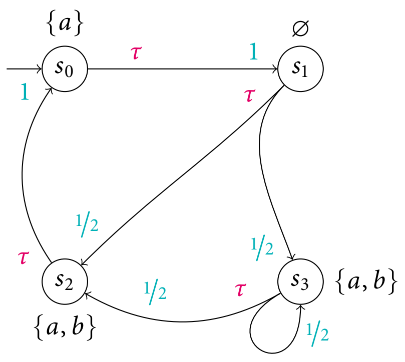


Figure 2.4: The example DTMC $\mathcal{D}$.

to variable valuations. We therefore, for instance, let

$$(\mathcal{S} \leftarrow \langle s_1 \rangle)(\mathfrak{s}) = \begin{cases} 0 & \text{if } \mathfrak{s} = \mathfrak{s}_1 \\ 1 & \text{if } \mathfrak{s} = \mathfrak{s}_0 \end{cases}$$

since the binary encoding of state index 1 is 01. With the mapping $x_0 = \mathfrak{s}_1, x_1 = \mathfrak{s}_0, x_2 = \mathfrak{s}_1'$ and $x_3 = \mathfrak{s}_0'$, we observe that the (reduced) MTBDD M' from Figure 2.3 encodes $\mathsf{M}_\mathbf{P}$. Reconsider the function $f_{\mathsf{M}'}$ represented by M' from Example 5. For instance, we have

$$f_{\mathsf{M}'}(\underbrace{0,1}_{s_1},\underbrace{1,1}_{s_3}) = \frac{1}{2} = \mathbf{P}(s_1, s_3)$$

to represent the transition from s_1 to s_3 with probability $1/2$.

Representing PA and MRA. Encoding nondeterministic models is naturally more complicated as the nondeterminism needs to be encoded as well. Not only do states have multiple available probability distributions, they may also possess several distributions labeled with the same action $\alpha \in \mathit{Act}$. In [Bai98], the author proposes to encode the nondeterminism using additional Boolean variables. This idea was taken up by [Par03] and worked out in detail. To the best of our knowledge, all probabilistic model checkers that support DD-based model checking of nondeterministic models use an encoding that follows this principle. Here, we only present the fundamentals of this approach and do not delve into its details.

We directly proceed to describe the encoding of an MRA $\mathcal{M} = \langle S, S^0, Act, \Delta, E, r, AP, L \rangle$. As argued in Section 2.2, MRA subsume PA and MA and the encodings for the latter can therefore be obtained by simply omitting information in a straightforward manner.

The elements S, S^0, L and E can be represented analogously to Markov chains (MCs). Furthermore, the state rewards r can be encoded using an MTBDD M_r with

$$\mathsf{M}_r(\mathcal{S} \leftarrow \langle s \rangle) = r(s)$$

To encode the nondeterminism in Δ, we extend the variable set to

$$Var_{\mathcal{M}} = \underbrace{\{\mathfrak{s}_1, \ldots, \mathfrak{s}_n\}}_{\mathcal{S}} \uplus \underbrace{\{\mathfrak{s}'_1, \ldots, \mathfrak{s}'_n\}}_{\mathcal{S}'} \uplus \underbrace{\{\mathfrak{a}_1, \ldots, \mathfrak{a}_k\}}_{\mathcal{A}} \uplus \underbrace{\{\mathfrak{n}_1, \ldots, \mathfrak{n}_\ell\}}_{\mathcal{N}}$$

where $n = \lceil \log_2 |S| \rceil$, $k = \lceil \log_2 |Act| \rceil$ and $\ell = \lceil \log_2 \max_s \max_\alpha |\{\langle s, \alpha, \rho, \mu \rangle\} \in \Delta| \rceil$. The variables in $\mathcal{A}$ are meant to encode the actions of the MRA and the variables in $\mathcal{N}$ are used to simply number the nondeterministic choices. Therefore, we extend the encoding $\langle \cdot \rangle$ twofold. First, we let $\langle \alpha \rangle$ map actions $\alpha \in Act$ to an encoding over the variables $\mathcal{A}$. For all actions in Act, this encoding needs to be distinct. Secondly, we use $\langle \langle s, \alpha, \rho, \mu \rangle \rangle$ to assign unique encodings over the variables $\mathcal{N}$ to choices $\langle s, \alpha, \rho, \mu \rangle \in \Delta$. We can then encode Δ through *two* MTBDDs M_Δ and M^ρ_Δ over Var. M_Δ encodes the transition probabilities without the rewards, whereas M^ρ_Δ encodes the rewards without the transition probabilities. Formally, they represent the functions

$$\mathsf{M}_\Delta(\mathcal{S} \leftarrow \langle s \rangle, \mathcal{A} \leftarrow \langle \alpha \rangle, \mathcal{N} \leftarrow \langle \langle s, \alpha, \rho, \mu \rangle \rangle, \mathcal{S}' \leftarrow \langle s' \rangle) = \begin{cases} \mu(s') & \text{if } \langle s, \alpha, \rho, \mu \rangle \in \Delta \\ 0 & \text{otherwise} \end{cases}$$

$$\mathsf{M}^\rho_\Delta(\mathcal{S} \leftarrow \langle s \rangle, \mathcal{A} \leftarrow \langle \alpha \rangle, \mathcal{N} \leftarrow \langle \langle s, \alpha, \rho, \mu \rangle \rangle, \mathcal{S}' \leftarrow \langle s' \rangle) = \begin{cases} \rho(s') & \text{if } \langle s, \alpha, \rho, \mu \rangle \in \Delta \\ 0 & \text{otherwise} \end{cases}$$

that together fully characterize Δ. Note that even though the representation is split, the connection between the probability distributions and their reward functions is not lost. For a choice $\langle s, \alpha, \rho, \mu \rangle \in \Delta$, the encodings $\langle \alpha \rangle$ and $\langle s, \alpha, \rho, \mu \rangle$ establish a unique link between a probability distribution and the reward function associated with it.

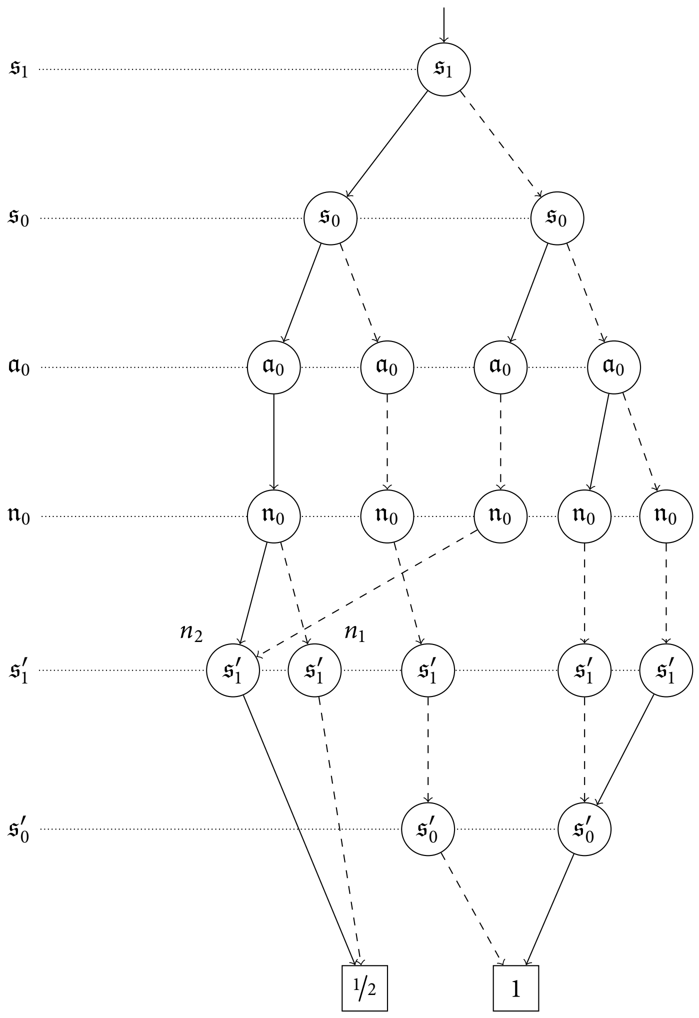


Figure 2.5: The MTBDD M_Δ.

Example 8. We now encode the MRA $\mathcal{M}$ from Example 1 in terms of DDs. Since $\mathcal{M}$ has four states, we need two Boolean state (unprimed) variables and let $\mathcal{S} = \{\mathfrak{s}_1, \mathfrak{s}_0\}$ (and $\mathcal{S}' = \{\mathfrak{s}'_1, \mathfrak{s}'_0\}$) . As in Example 7, we choose the canonical mapping of state indices to their binary representation to encode the states. As $\mathcal{M}$ has two actions, namely $Act = \{\tau, \Lambda\}$, it suffices to have one action variable $\mathcal{A} = \{\mathfrak{a}_0\}$ and we map Λ to the valuation which assigns 0 to $\mathfrak{a}_0$ and τ to the valuation mapping $\mathfrak{a}_0$ to 1. Finally, to encode the nondeterministic choice between the two choice labeled with τ in s_3, we need one more variable $\mathcal{N} = \{\mathfrak{n}_0\}$. To improve the readability of the MTBDDs, we choose the variable ordering

$$\mathfrak{s}_1 \prec \mathfrak{s}_0 \prec \mathfrak{a}_0 \prec \mathfrak{n}_0 \prec \mathfrak{s}'_1 \prec \mathfrak{s}'_0$$

instead of an interleaved one that would be preferable in practice.

Figure 2.5 shows the M_Δ representing the transition relation Δ of $\mathcal{M}$. It uses the action variables $\mathcal{A}$ to distinguish Markovian from probabilistic choices and the nondeterminism variables $\mathcal{N}$ to distinguish nondeterministic (probabilistic) choices with the same label. In the case of $\mathcal{M}$, we only need the variable $\mathfrak{n}_0$ to distinguish the two choices in s_3. For instance, we have that

$$f_{\mathsf{M}_\Delta}(\underbrace{\overbrace{1,1}^{\mathfrak{s}_1,\mathfrak{s}_0}}_{s_3}, \underbrace{\overbrace{1}^{\mathfrak{a}_0}}_{\tau}, \overbrace{0}^{\mathfrak{n}_0}, \overbrace{\cdot,\cdot}^{\mathfrak{s}'_1,\mathfrak{s}'_0})$$

is represented by the node n_1 in Figure 2.5 and corresponds to the distribution $\mu_{3,1}$ (from Example 1) available in s_3 whereas

$$f_{\mathsf{M}_\Delta}(\underbrace{\overbrace{1,1}^{\mathfrak{s}_1,\mathfrak{s}_0}}_{s_3}, \underbrace{\overbrace{1}^{\mathfrak{a}_0}}_{\tau}, \overbrace{1}^{\mathfrak{n}_0}, \overbrace{\cdot,\cdot}^{\mathfrak{s}'_1,\mathfrak{s}'_0})$$

is represented by n_2 and corresponds to distribution $\mu_{3,0}$.

In Figure 2.6 we show the transition reward MTBDD M^ρ_Δ. Since it uses the same variables $\mathcal{A} \uplus \mathcal{N}$ to encode action labels and nondeterministic choices, the connection between the distributions of choices and their reward function is retained.

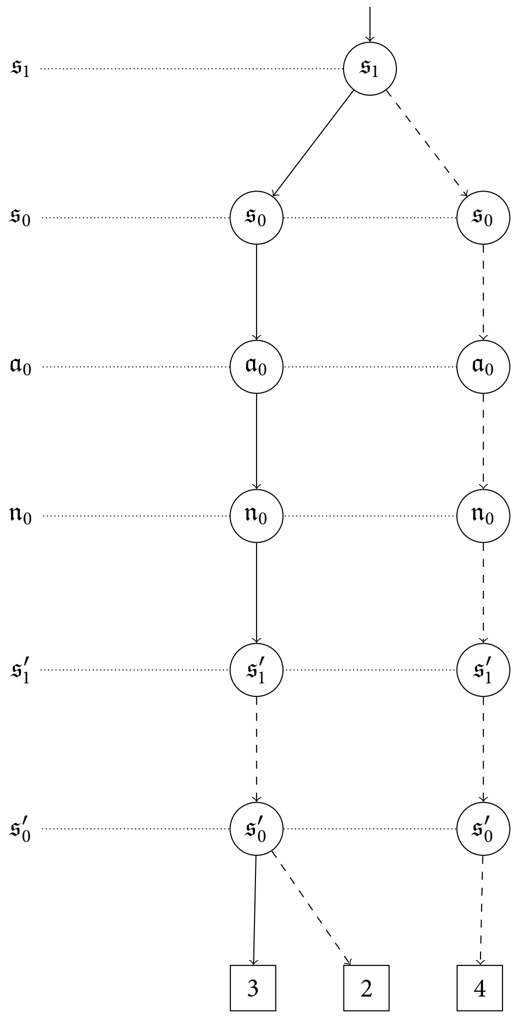


Figure 2.6: The transition reward MTBDD $\mathsf{M}^{\rho}_{\Delta}$.

2.6 Solving

Many problems arising in verification reduce to solving fundamental underlying problems. In the context of the (automated) analysis of probabilistic systems, probably the most common problem is that of solving a set of linear equations. They occur, for example, in the context of computing (unbounded) reachability probabilities for DTMCs and CTMCs. For models involving nondeterminism, Bellman equations [Bel58] have to be solved. One of the two main methods for this is policy iteration [How64], which reduces the problem to solving a series of linear equation systems. Here, we want to mention two additional problem formulations that play a role in the course of the thesis.

Mixed-Integer Linear Programming Let *Var* be a set of integer- and real-valued variables. Intuitively, a mixed-integer linear program is a set of linear inequalities over *Var* together with a linear objective function. The goal is to find a valuation $v \in Val(Var)$ that satisfies all inequalities and produces an objective value that is maximal (or minimal) among all solutions.

Every linear program (LP) is also an MILP. In fact, the most common approach to solving MILPs is to relax the problem to an LP and then successively bound the values of the integer variables that have a non-integral value in the solution to the relaxed problem. Well-known solvers for MILPs include the free tools GLPK[1] and LPSOLVE[2] as well as the commercial tools CPLEX[3] and GUROBI[4].

Satisfiability Modulo Theories One the most fundamental problems in computer science is the satisfiability (SAT) problem. Given a formula $\varphi \in Bxp(Var)$ over a set of Boolean variables that uses only the regular Boolean operators, the question is whether there exists a variable valuation $v \in Val(Var)$ such that $v \models \varphi$. While the problem is NP-complete, actual solvers have made substantial progress in the past decades and scale to problems with a huge number of variables[5].

Satisfiability modulo theories (SMT) generalizes the regular satisfiability problem to other theories. More specifically, the variables do not necessarily have Boolean type, but may be integer, real or even more complex types such as strings. The expressions then may involve not only the Boolean connectives but also, for instance, arithmetical operations or comparisons between numeric variables. However, one has to be careful

[1]`https://www.gnu.org/software/glpk/`
[2]`http://lpsolve.sourceforge.net/5.5/`
[3]`https://www.ibm.com/analytics/cplex-optimizer`
[4]`http://www.gurobi.com/`
[5]`http://www.satcompetition.org/`

which operations and variable types are allowed precisely, because the decision problem is undecidable for the more expressive theories. For example, in general it is not possible to decide whether a satisfying valuation exists if the expression involves integer-valued variables and non-linear arithmetic (also referred to as Peano arithmetic) [Göd31]. However, other important theories such as the linear theory of the integers (*Presburger arithmetic* or linear integer arithmetic (LIA)) and the non-linear theory of the reals (linear real arithmetic (LRA)) are decidable and extensive tool support exists[6].

Virtually all SAT (and SMT) solvers achieve this by applying the DPLL algorithm (and DPLL(T)) [DP60; DLL62; Tin02; Gan+04] and conflict-driven clause learning [SS96]. Briefly speaking, the core idea of the approach is to assign truth values to free variables as long as possible. Upon reaching a point where the formula cannot be satisfied under the current (partial) valuation (a “conflict”), a formula is learned that explains the conflict. On the logic level, this formula is implied by the original problem and therefore also called a *lemma*. Then a backtracking step happens that revises one of the earlier assignment decisions. As the lemmas are still implied if the original problem is *extended*, solvers work incrementally in the sense that the previously learned knowledge about the formulae can speed up future calls significantly. To optimize information reuse, some solvers go so far as to provide dedicated methods for the $\textsc{AllSat}(\Phi)$ problem, where the goal is to find *all* satisfying valuations of a set of formulae Φ.

Modern solvers go well beyond answering the satisfiability problems. For instance, if the solver finds a problem to be unsatisfiable, it may offer to extract an *unsatisfiable core*, which is a (typically) small “subformula” of the original formula that already explains the unsatisfiability. Using unsatisfiable cores or counter circuits [FM06; Si+16] the MaxSat (or dually the MinSat) problem can be solved. Here, for two (satisfiable) sets of formulae Φ_1 and Φ_2, the goal is to determine a variable valuation $v \in \mathit{Val}(\mathit{Var})$ that satisfies all formulae in Φ_2 and simultaneously satisfies a maximal (or minimal) number of formulae of Φ_1. Formally, with $\mathit{Sol}(\Phi) = \{v \mid \forall \varphi \in \Phi \,.\, v \models \varphi\}$ and $\Phi^v = \{\varphi \in \Phi \mid v \models \varphi\}$, we let

$$\textsc{MaxSat}(\Phi_1, \Phi_2) \in \left\{v \in \mathit{Sol}(\Phi_2) \mid \forall v' \in \mathit{Sol}(\Phi_2)\,.\,|\Phi_1^{v}| \geq |\Phi_1^{v'}|\right\}, \text{ and}$$
$$\textsc{MinSat}(\Phi_1, \Phi_2) \in \left\{v \in \mathit{Sol}(\Phi_2) \mid \forall v' \in \mathit{Sol}(\Phi_2)\,.\,|\Phi_1^{v'}| \geq |\Phi_1^{v}|\right\}.$$

Finally, some solvers, e. g. MathSat, provide methods to synthesize *interpolants*. Let $\langle \varphi_1, \varphi_2 \rangle$ be a pair of logical formulae $\langle \varphi_1, \varphi_2 \rangle$ whose conjunction $\varphi_1 \wedge \varphi_2$ is unsatisfiable. An interpolant for $\langle \varphi_1, \varphi_2 \rangle$ is a formula φ over the variables $\mathit{Var}(\varphi_1) \cap \mathit{Var}(\varphi_2)$ with

$$\varphi_1 \implies \varphi \quad \text{and} \quad \varphi_2 \wedge \varphi \text{ is unsatisfiable.}$$

[6] `http://smtcomp.sourceforge.net/`

Intuitively, an interpolant succinctly captures the essence of why the conjunction of two formulae is unsatisfiable. Interpolants are very successfully applied in regular model checking [McM18].

Chapter 3

The JANI Modeling Language

3.1 Motivation and Goals

As probabilistic model checking became increasingly mature in the past decades [Kat16], more and more tools were developed that support the automated analysis of systems involving probabilistic aspects. While these tools can handle more complex model types, larger state spaces and more properties than ever before, little effort was invested in developing the tools' input languages. This is not very surprising as most of these tools are developed in an academic context that tends to focus more on features than usability. As most tools have been tailored to solve a very specific task at hand, they either define their own input language or accept a dialect — typically an extended subset — of another tool's input language. The latter is done in an attempt to not deviate too far from existing languages but also to save development effort as infrastructure like parsers and data structures are tedious and error-prone to develop from scratch but easy to reuse. This has led to a scattered language landscape in which it is rarely the case that the same input model can be treated by several tools.

This severely hinders the adoption of probabilistic model checking by, e. g., industry, as potential users have to commit to a specific tool and its input language and, for this, have to have a deeper understanding of the capabilities and characteristics of the tools. As developing complex models is still a demanding and time-consuming task, having to commit to a single tool and its features and restrictions potentially exceeds the risks one is willing to take. Furthermore, the large number of modeling languages slows down research significantly: state-of-the-art techniques tend to be sophisticated and establish results through the use of a variety of different algorithms and data structures.

In practice these ingredients may happen to be spread over several tools. If these tools are not able to refer to the very same model, this even prevents simple proof-of-concept implementations unless the researchers conduct a time-consuming reimplementation of the missing parts in one of the tools or perform non-trivial transformations between the input languages. Finally, it impairs the reuse of benchmark models and therefore hinders an effective comparison of tools.

We seek to improve the current situation by introducing *another* language, called **JANI**. Obviously[1], introducing a new language is associated with the risk of dividing the current landscape further and having yet another language. To mitigate that risk, we argue that **JANI** has been designed with the following key requirements in mind:

(I) easy to integrate in new and existing tools,

(II) extensibility to make the language future-proof,

(III) general enough to capture a wide range of existing modeling formalisms,

(IV) succinct and symbolic representation of huge (or even infinite) state spaces,

(V) rigorously defined semantics.

When presenting the **JANI** language, we refer to these requirements and show how they are fulfilled. We briefly mention that the **JANI** language actually consists of two components. **jani-model** defines a format to specify models involving quantitative aspects meant to be supported as input language by several tools. Complementing this, **jani-interaction** provides a communication protocol with which tools can communicate models, tasks, features and much more. In the further course of this thesis, we focus entirely on **jani-model** and therefore, for simplicity, use **JANI** and **jani-model** interchangeably. For further details on **jani-interaction**, we refer to [Bud+17]. To distinguish between languages and the (sometimes identically named) tools, we will use bold fonts for **languages** and small capitals for TOOLS.

3.2 Syntax

JANI models are encoded in the **json** data format. **json** is a text-based, human-readable, lightweight data-interchange format [Bra14]. Unlike alternatives such as XML [96], it is extremely simple: in fact, its complete grammar can be captured by five small syntax diagrams[2]. Due to its simplicity, there exist generic **json**-parsers for virtually all

[1] see `https://xkcd.com/927/`

[2] `https://www.json.org`

programming languages. For tool developers this means that they do not have to write code that is concerned with parsing the input language. Instead, it suffices to transform the **json** input tree to suitable data structures. In **JANI**, this is further supported by other design decisions such as – in contrast to other modeling languages – encoding expressions as trees rather than flat strings to eliminate the need for an expression parser.

Besides getting the parser "for free", **json** provides the flexibility to extend existing specifications without breaking compatibility with tools that accept only the subset that was the original specification. In addition, **JANI** specifies versioning constructs that allow to deal with future amendments to previously defined semantics if there should be the need to. Again, this allows for future extensions without breaking compatibility with tools that only support a previous version as **json** allows to ignore unknown attributes.

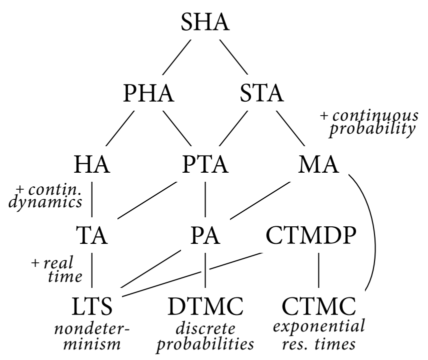
Finally, choosing **json** has another key advantage: By providing a grammar (also referred to as *schema*), *schema validators* such as Js-Schema[3] allow to rigorously validate whether a given **json** input is contained in the language and is to be accepted. We therefore decided to formalize the grammar of **JANI** in terms of a Js-Schema schema[4]. This schema covers most syntactical checks of **JANI** models, but some more complex ones like the existence of a certain attribute if and only if another one is present, cannot be expressed in Js-Schema. However, the schema documents these additional requirements in the form of verbose annotations. In summary, relying on **json** already fulfills requirements (II) and (at least partially) (I).

At this point, we abstain from giving the formal grammar of **JANI**. Instead we proceed with the semantics of the key modeling elements and, by example, highlight how these are expressed syntactically.

3.3 Semantics

By design, **JANI** is intended to be able to capture a broad class of modeling formalisms from the quantitative verification world. Figure 3.1 illustrates which model types are supported by **jani-model** as well as their interconnections. The most basic models are labeled transition systems (LTS), discrete-time Markov chains (DTMC) and continuous-time Markov chains (CTMC). At the other end of the spectrum, the most general model are stochastic hybrid automata (SHA, [Frä+11]) that combine the features of probabilistic hybrid automata (PHA, [Spr00]) and stochastic timed automata (STA, [Boh+06]).

[3]`https://github.com/molnarg/js-schema`
[4]available at www.jani-spec.org/



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
PA	probabilistic automata
CTMDP	continuous-time MDPs
LTS	labelled transition systems
DTMC	discrete-time Markov chains
CTMC	contin.-time Markov chains

Figure 3.1: Model types supported by the **jani-model** format [Bud+17].

Together with the embeddings from several well-known modeling languages detailed in Section 3.4, **JANI** therefore satisfies requirement (III) and — as pointed out earlier — provides natural extension mechanisms to potentially cover even more model types in the future (requirement (II)).

In this thesis, we focus on models that *do not* involve hybrid aspects like the notion of time in timed automata or the differential equations in hybrid modeling formalisms. More concretely, we focus on the subclass of **jani-model** that encode Markov automata (and therefore also labelled transition systems (LTSs), DTMCs, CTMCs and PA).

3.3.1 Symbolic Markov Automata

Symbolic modeling languages, such as **PRISM** [KNP11], **Modest** [Boh+06] and **Uppaal** [Beh+06], build on a set of symbolic variables that implicitly span the state space as the Cartesian product of the domains of these variables. Encoding states like this allows to succinctly represent enormous or even infinite state spaces, in particular for systems that tend to be highly structured. In contrast to an explicit enumeration of all states and transitions, using symbolic variables and, consequently, a symbolic representation of the transitions also makes models more readable and the modeling process less error-prone. **JANI** also adopts symbolic variables to satisfy criterion (IV).

For our presentation, we assume a finite set of typed variables *Var* where each variable

$x \in Var$ has (potentially infinite) domain $Dom(x)$. We introduce *assignments* that are used to symbolically encode updates to the variables in Var as follows.

Definition 15 (Assignment)**.** An *assignment* a for a set of variables Var is a partial function

$$a\colon Var \rightharpoonup Exp(Var) \cup \{\top\}$$

that (i) denotes an over-specification of variable x by $a(x) = \top$ and (ii) respects the types of the variables in the sense that the expression $a(x) \neq \top$ always evaluates to an element in $Dom(x)$, i. e.,

$$a(x)@v \in Dom(x) \text{ for all } v \in Val(Var).$$

We use $Asg(Var)$ to refer to the set of assignments over Var. Let $a_\perp$ with $a_\perp(x) =\perp$ for all $x \in Var$ be the empty assignment. For $x \in Var$ and $\eta \in Exp(Var)$, we let $a[x' \mapsto e]$ be the assignment that mimics a but maps x' to η instead of $a(x')$, i. e.

$$a[x' \mapsto e](x) = \begin{cases} a(x) & \text{if } x \neq x' \\ e & \text{otherwise.} \end{cases}$$

For convenience, we write

$$x_1' = \eta_1, \dots, x_n' = \eta_n$$

to denote the assignment with

$$a(x) = \begin{cases} \eta_i & \text{if } x = x_i \\ \perp & \text{otherwise.} \end{cases}$$

Note that assignments are partial functions and we use $a(x) =\perp$ to indicate that a does not assign to x. Similarly, $a(x) = \top$ is used to indicate that the assignment *over-specifies* what is being assigned to x, an error that can occur when composing multiple automata. As assignments map variables to expressions that symbolically represent their "new" value, they encode a variable valuation transformer.

Definition 16 (Semantics of an Assignment)**.** The *semantics of an assignment* $a \in Asg(Var)$ with $a(x) \neq \top$ for all $x \in Var$ is the variable valuation transformer given by

$$\llbracket a \rrbracket \colon Val(Var) \to Val(Var)$$

$$\llbracket a \rrbracket(v)(x) = \begin{cases} a(x)@v & \text{if } a(x) \neq \perp \\ v(x) & \text{otherwise} \end{cases}$$

Intuitively, $\llbracket a \rrbracket$ *simultaneously* updates all variables x with $a(x) \neq \perp$ according to the value of $a(x)$ in v and preserves the values of the remaining variables. As a next step, we introduce *indexed assignments* that chain several assignments.

Definition 17 (Indexed Assignment)**.** An *indexed assignment* for a set of typed variables Var is a function

$$ia \colon \mathbb{Z} \to Asg(Var)$$

such that there exist indices $i, j \in \mathbb{Z}$ with

$$\forall k \,.\, k < i \vee k > j \implies ia(k) = a_\perp.$$

We use $IAsg(Var)$ to denote the set of indexed assignments over Var and let $ia_\perp$ be the empty indexed assignment given by

$$ia_\perp(i) = a_\perp \text{ for all } i \in \mathbb{Z}.$$

For an indexed assignment $ia \neq ia_\perp$, we use $\min(ia)$ and $\max(ia)$ to denote the smallest and largest indices i, respectively, for which $ia(i) \neq a_\perp$. For convenience, we set $\min(ia_\perp) = \max(ia_\perp) = 0$. We call an indexed assignment ia *simple* if $ia(i) = a_\perp$ for all $i \neq 0$. For convenience, for ordered indices $i_1 < \ldots < i_n$, we write

$$ia = i_1 \colon a_{i_1} \fatsemi \ldots \fatsemi i_n \colon a_{i_n}$$

to denote the indexed assignment with

$$ia(i) = \begin{cases} a_{i_k} & \text{if } i = i_k \text{ for some } 1 \leq k \leq n \\ a_\perp & \text{otherwise.} \end{cases}$$

Similar to assignments, we use $ia_{i \mapsto a}$ to denote the indexed assignment that mimics ia but applies a at index i instead of $ia(i)$.

An indexed assignment ia provides a sequence of assignments for the (finite) range of indices between $\min(ia)$ and $\min(ia)$ with the interpretation that first the assignment at index $\min(ia)$ is carried out, then the assignment at index $\max(ia) + 1$, and so on. Therefore, just like for regular assignments, the semantics of an indexed assignment is a valuation transformer.

Definition 18 (Semantics of an Indexed Assignment)**.** For an indexed assignment $ia \in IAsg(Var)$, its semantics is the variable valuation transformer

$$\begin{aligned} &[\![ia]\!] \colon Val(Var) \to Val(Var) \\ &[\![ia]\!](v) = \left([\![a_{\max(ia)}]\!] \circ \ldots \circ [\![a_{\min(ia)}]\!]\right)(v) \end{aligned}$$

Hence, the semantics of an indexed assignment ia corresponds to the sequential application of the assignments for increasing indices. Using indexed assignments, we now formally capture a *symbolic probability distribution*.

Definition 19 (Symbolic Probability Distribution)**.** A *symbolic probability distribution* over a set of variables Var and a finite set of locations Loc is a function

$$D \colon IAsg(Var) \times Loc \to Qxp(Var)$$

such that

(i) there are only finitely many $\langle ia, \ell \rangle$ with $D(ia, \ell) \neq 0$ where 0 is the constant zero expression, and

(ii) $0 \leq D(ia, \ell)@v \leq 1$ for all $ia \in IAsg(Var)$, $\ell \in Loc$, $v \in Val(Var)$,

(iii) for all $v \in Val(Var)$ it is $\sum_{ia \in IAsg(Var), \ell \in Loc} D(ia, \ell)@v = 1$

$SDist(Var, Loc)$ is used to denote the set of symbolic probability distributions over Var and Loc. As for regular probability distributions, we define the support of a symbolic probability distribution D as

$$supp(D) = \{\langle ia, \ell \rangle \mid D(ia, \ell) \neq 0\}\,.$$

For convenience, we abbreviate $|supp(D)|$ by $|D|$.

A symbolic probability distribution D maps a tuple $\langle ia, \ell \rangle$ to an expression $q = D(ia, \ell)$. The indexed assignment (symbolically) describes the updates to the variables and ℓ describes the location update of the symbolic automaton. Then, q describes the probability of the occurrence of the update $\langle ia, \ell \rangle$. We will now formally define symbolic Markov automata (SMA), the main element of a **JANI** specification.

Definition 20 (Symbolic Markov Automaton)**.** A *symbolic Markov automaton (SMA)* is a tuple

$$\mathfrak{A} = \left\langle Loc, Var = PV \uplus TV, v_{TV}, TL, Act, \ell^0, Init^0, E \right\rangle$$

consisting of a

» a finite set of locations Loc,

» a finite set of typed variables Var, partitioned into a set of *permanent* variables PV and a set of *transient* variables TV,

» a default transient variable valuation $v_{TV} \in Val(TV)$,

» a transient location assignment $TL\colon Loc \to Asg(TV)$,

» a finite set of actions Act with $\tau \in Act$,

» an initial location $\ell^0 \in Loc$,

» an initial condition $Init^0 \in Bxp(PV)$

» an edge function

$$E\colon Loc \to \mathscr{P}\left(Bxp(Var) \times (Act \uplus Qxp(Var)) \times SDist(Var, Loc)\right)$$

that assigns to each location a set of edges. For an edge $\langle g, \alpha, D\rangle \in E(\ell)$, we call $e' = \langle \ell, g, \alpha, D\rangle$ an (location-extended) edge and also write $e' \in E$. Furthermore, we refer to

- ℓ as the *source location* of e (written $src(e)$),
- g as the *guard of* e (written $g(e)$),
- α as the *action of* e (written $\alpha(e)$) if $\alpha \in Act$ or the *rate of* e (written $\lambda(e)$) if $\alpha \in Qxp(PV)$, and
- D as the *distribution* of e (written $D(e)$)

If $\alpha = \lambda \in Qxp(PV)$, we call e a *Markovian edge* and a *probabilistic edge* otherwise. Let $E^M_{\mathfrak{A}}$ and $E^P_{\mathfrak{A}}$ denote the set of Markovian and probabilistic edges in $\mathfrak{A}$, respectively, and $E^M_{\mathfrak{A}}(\ell) = E^M_{\mathfrak{A}} \cap E(\ell)$. A symbolic Markov automaton $\mathfrak{A}$ with $E^M_{\mathfrak{A}} = \varnothing$ is called a *symbolic probabilistic automaton* (SPA). If the indexed assignments of all edges of a symbolic Markov automaton are simple, we call the overall automaton simple.

The automaton $\mathfrak{A}$ starts is in its initial location ℓ^0 with an initial variable valuation that satisfies its initial condition $Init^0$. The edges of the automaton $\mathfrak{A}$ connect the locations via their symbolic probability distributions. In addition to symbolically specifying a probability for a location update, they can update the variables of the automaton through the corresponding indexed assignments. Edges labeled with actions α from Act can be used to interact with the environment ($\alpha \neq \tau$) or express that the behavior is internal to the automaton ($\alpha = \tau$). If the edge specifies a rate instead of a guard, it is a Markovian edge and expresses that time may be spent in the source location before taking the edge similar to the Markovian choices in MRA.

In contrast to simpler modeling formalisms, SMA distinguish *permanent* and *transient* variables. Unlike permanent variables, transient variables have no "history" and can be viewed as being periodically reset. Unless otherwise specified by the automaton, they always have their default value prescribed by v_{TV} whenever they are read. They can, however, be explicitly set in a location ℓ by the transient location assignment $TL(\ell)$ and along edges in the form of (indexed) assignments. Intuitively, upon taking an edge, transient variables are

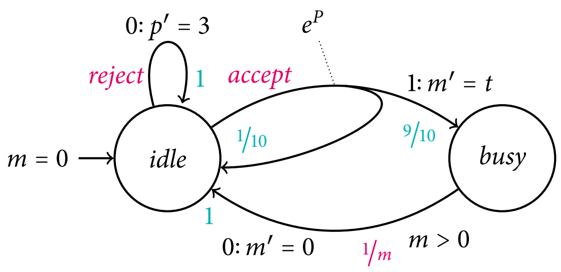


Figure 3.2: An example SMA $\mathfrak{A}_R$.

(i) first initialized with their default value,

(ii) then potentially read and written by the indexed assignments, and finally

(iii) reset according to $TL(\ell')$ or to their default value if $TL(\ell') = \perp$ upon entering the target location ℓ'.

The formal semantics of SMA will be made more precise in Section 3.3.3 on page 59.

Example 9. Consider the symbolic Markov automaton

$$\mathfrak{A}_R = \left\langle Loc_R, Var_R = PV_R \uplus TV_R, \nu_{TV,R}, TL_R, Act_R, \ell^0_R, Init^0_R, E_R \right\rangle$$

depicted in Figure 3.2. Let us go through the formal elements of $\mathfrak{A}_R$. The location set of the automaton $Loc_R = \{idle, busy\}$ consists of two locations. *idle* is the initial location ℓ^0_R and is graphically represented by the incoming arrow without a source, along which the initial condition $Init^0 = (m = 0)$ can be read off. In total, the automaton uses three variables: $PV_R = \{m\}$ is permanent and the other two $TV_R = \{t, p\}$ are transient. In general, the default values for the transient variables are given by 0 unless explicitly stated otherwise. We represent transient location assignments within the locations. However, $\mathfrak{A}_R$ has $TL(\ell) = a_\perp$ for all locations ℓ. The automaton possesses three edges over the action set $Act_R = \{reject, accept, \tau\}$. Consider the edge $e^P \in E_R(idle)$ with $\alpha(e^P) = accept$. Its guard is the expression *true* and, unless explicitly stated otherwise, we omit the guard in this case for better readability. The support $supp(D(e^P))$ of its symbolic probability distribution has two elements:

$D(e^P)(idle, a_\perp) = 1/10$ and $D(e_1^P)(idle, 1{:}\, a_\perp[m \mapsto t]) = 9/10$. To further improve readability, we will often omit the indices of simple assignments.

The automaton models a receiver that can be either *idle* or *busy*. The intuition behind its variables is as follows: m stores a message that is received whereas t is used as a means of receiving the message and p measures a penalty that is gathered in certain scenarios. When a message is received while the receiver is *idle*, the $\mathfrak{A}_R$ can nondeterministically choose to either *reject* it and remain in *idle* or *accept* it and become busy processing the message. In the former case, it accumulates a penalty of 3. In the latter case, the message is accidentally dropped with a probability of 0.1 and takes the system into *busy* with probability 0.9. The automaton expects the message it is receiving to be stored in variable t. For this, it allows the sender to write the message to t at index 0 and then goes on to read the content of the variable and store it into its local variable m at index 1. In location *busy*, the automaton needs an exponentially distributed time depending on the message m it received to process m. More specifically, the higher the value of the message m, the longer the receiver needs to process it.

Note how the automaton uses transient variables for two different scenarios. First, it uses t to read a message from other automata in a network, a concept that we detail later. And secondly, p is used to accumulate a penalty for certain behaviors, which can be leveraged to derive reward models. In both cases, the history of these variables is irrelevant and they are therefore marked as transient.

Figure 3.3 shows an excerpt how the SMA $\mathfrak{A}_R$ is syntactically represented in **JANI**. The full encoding can be found in Appendix A.

We want to highlight the advantages of using location-based automata as opposed to location-less formalisms. While locations are syntactic sugar and could be expressed in terms of dedicated variables of the automaton, making them explicit provides a natural entity to which invariants in the timed and hybrid models can be attached. Furthermore, as many models represent algorithms or processes that follow a regular structure, encoding their control locations via "program counter" variables mixes data and control variables. Keeping them separate in **JANI** provides more structural information, which potentially eases tasks such as model checking or static analysis.

3.3.2 Communication and Composition

In order to capture large, yet structured, systems succinctly, most modeling formalisms provide a means to compose basic modeling entities. For example, **PRISM** features

```
{
    "name": "$\mathfrak{A}_R$",
    "locations": [ { "name": "idle" }, { "name": "busy" } ],
    "initial-locations": [ "idle" ],
    "variables": [ { "name": "m", "type": "int" } ],
    "restrict-initial": { "exp": { "left": "m", "op": "=", "right": 0 }
         },
    "edges": [
        {
            "location": "idle",
            "action": "reject",
            "destinations": [
                {   "assignments": [ { "ref": "p", "value": "3" } ],
                    "location": "idle"  }
            ]
        },
        {
            "location": "idle",
            "action": "accept",
            "destinations": [
                {   "probability": { "exp": { "left": 9, "op": "/", "
                    right": 10 } },
                    "assignments": [ { "ref": "m", "value": "t", "index
                        ": 1 } ],
                    "location": "busy"  },
                {   "probability": { "exp": { "left": 1, "op": "/", "
                    right": 10 } },
                    "location": "idle"  }
            ]
        },
        {
            "location": "busy",
            "guard": { "exp": { "left": "m", "op": ">", "right": 0 } },
            "rate": { "exp": { "left": 1, "op": "/", "right": "m" } },
            "destinations": [
                {   "assignments": [ { "ref": "m", "value": 0 } ],
                    "location": "idle"  }
            ]
        }
    ]
}
```

Figure 3.3: An excerpt from the **json** representation of $\mathfrak{A}_R$.

a parallel composition of its reactive modules and **Uppaal** composes symbolic timed automata. Despite being syntactically different, **JANI** leverages similar composition mechanisms through the use of networks of symbolic Markov automata, which further contributes to support requirement (IV).

Intuitively, the automata contained in a network execute in parallel. They may perform steps via edges either asynchronously or synchronously. In the former case, one automaton moves according to the probabilities and updates specified by the symbolic probability distribution of an edge. In the latter case, several automata need to synchronize over edges and move *simultaneously*.

The concept of parallel composition is well-known and well-understood, helping **JANI** to adhere to requirement (V). Formally, a network of symbolic Markov automata is defined as follows.

Definition 21 (Network of Symbolic Markov Automata)**.** A (parallel) *network of symbolic Markov automata (NSMA)* is a tuple

$$\mathfrak{N} = \langle \mathfrak{A}_1, \ldots, \mathfrak{A}_n, Syn \rangle$$

comprising

» $n \geq 1$ symbolic Markov automata

$$\mathfrak{A}_i = \left\langle Loc_i, Var_i = PV_i \uplus TV_i, \nu_{TV,i}, TL_i, Act_i, \ell_i^0, Init_i^0, E_i \right\rangle$$

such that for $1 \leq i, j \leq n$, we have $PV_i \cap TV_j = \varnothing$,

» a set $Syn \subseteq Act_1^{\perp} \times \ldots \times Act_n^{\perp} \times Act(\mathfrak{N})$ of *synchronization vectors* where

- $Act_i^{\perp} = (Act_i \setminus \{\tau\}) \uplus \{\perp\}$, and
- $Act(\mathfrak{N}) = \bigcup_{i=1}^{n} Act_i$.

We use $Var(\mathfrak{N}) = \bigcup_{i=1}^{n} Var_i$ to denote the set of variables of the network and let

$$E(\mathfrak{N}) = \{\langle \ell, g, \alpha, D \rangle \mid \langle \ell, g, \alpha, D \rangle \in E_i(\ell) \text{ for some } 1 \leq i \leq n\}$$

be the (location-extended) edges of the network. If all automata $\mathfrak{A}_1, \dots, \mathfrak{A}_n$ are symbolic probabilistic automata, the network is called a network of symbolic probabilistic automata (NSPA).

Technically, within a **JANI** specification, variables need to be explicitly declared as global to allow sharing them between several automata. This is in contrast to, e. g., **PRISM**'s reactive modules where a module can read (but not write) other modules' variables by default and there is no notion of private or unobservable behavior between modules. In our formal presentation, we will abstract from this and assume that all variables that are used by more than one automaton are marked as being global.

All automata that are part of the network have access to these shared variables in guards and variable assignments along edges. However, all other variables are private to the individual automata and cannot be read or written by other automata of the network. Explicitly stating the scope of visibility promotes encapsulation and makes coupling between automata more visible.

Unlike the process algebra synchronization available in **PRISM**, **JANI** draws inspiration from CADP's **Exp 2.0** language [INR] and offers so-called *synchronization vectors*. In essence, a synchronization vector $v = \langle \alpha_1, \dots, \alpha_n, \alpha \rangle$ specifies a subset of the automata of the network, namely all $\mathfrak{A}_i$ for which $\alpha_i \neq \perp$. The meaning of v then is that the overall system may perform an α action if all selected automata perform the selected actions $\alpha_i \neq \perp$ *synchronously*. In this context, "synchronously" means that all affected automata update the variables according to the participating edges' (indexed) assignments. All other automata do not take part in the handshaking and retain their state. Note that τ is assumed to express an internal action of an automaton and can therefore not be used for synchronization.

Example 10. Recall the SMA $\mathfrak{A}_R$ from Example 9. The SMA $\mathfrak{A}_S$ depicted in Figure 3.4 models a station that sends messages to $\mathfrak{A}_R$. The sender generates messages with exponentially distributed delay with mean $\lambda \in \mathbb{R}_{>0}$. In location *send* it writes the message 1 to the shared (and therefore global) variable t at index 0 with a probability of $2/3$. In the remaining cases, it first writes the message 2 to t and incurs a penalty using the (global) penalty variable p at index 1. Furthermore, the automaton accumulates a penalty of 1 when in location *send*, which is modeled using the transient location assignment denoted within the location. We make this more precise in Section 3.3.5

As the overall system is meant to send and receive the messages synchronously, we have to provide appropriate synchronization vectors and therefore consider the

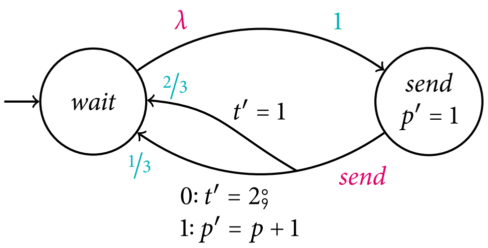


Figure 3.4: An SMA $\mathfrak{A}_S$ modeling a sending station.

NSMA $\langle \mathfrak{A}_S, \mathfrak{A}_R, \{v_1, v_2\}\rangle$ with $v_1 = \langle send, accept, \tau\rangle$ and $v_2 = \langle send, reject, \tau\rangle$. The two synchronization vectors enforce that (i) a message can only be send if the receiver is *idle* and therefore ready to handle a message, (ii) that a message can only be accepted if the sender is in mode *send* and (iii) that sending and receiving happens simultaneously. The automaton $\mathfrak{A}_S$ as well as the synchronization are also encoded in the **JANI** model in Appendix A.

Now that we have introduced NSMA, it remains to make the semantics of an NSMA more precise. We do this by defining a single equivalent SMA. As we need to compose edges of individual automata, we first define the composition of assignments and symbolic probability distributions, which are the basic building blocks of edges.

Definition 22 (Composition of (Indexed) Assignments). Let Var_1 and Var_2 be sets of typed variables and $Var = Var_1 \cup Var_2$.

(i) Let $a_1 \in Asg(Var_1)$, $a_2 \in Asg(Var_2)$. The composition $a_1 \otimes a_2 \in Asg(Var)$ is given by

$$(a_1 \otimes a_2)(x) = \begin{cases} a_1(x) & \text{if } a_1(x) \neq \top \wedge a_2(x) = \perp \\ a_2(x) & \text{if } a_2(x) \neq \top \wedge a_1(x) = \perp \\ \top & \text{otherwise} \end{cases}$$

(ii) Let $ia_1 \in IAsg(Var_1)$, $ia_2 \in IAsg(Var_2)$. The composition $ia_1 \otimes ia_2 \in$

$\mathit{IAsg}(\mathit{Var})$ is given by

$$(ia_1 \otimes ia_2)(i) = ia_1(i) \otimes ia_2(i) \text{ for all } i \in \mathbb{Z}.$$

Intuitively, the composition of two assignments preserves the mappings of its input assignments as long as only one of them actually assigns the variable in question. If both assign a value to the same variable, the result is marked as overdefined, represented by $\top$. For indexed assignments, the assignments at the individual indices are composed to create the composed indexed assignment.

Example 11. Consider the assignments $a_1 = a_\bot[p \mapsto p+1]$ and $a_2 = a_\bot[m \mapsto t]$ from the automata $\mathfrak{A}_S$ from Example 10 and $\mathfrak{A}_R$ from Example 9. The composition $a_1 \otimes a_2$ is the assignment $a_\bot[p \mapsto p+1][m \mapsto t]$. Now, consider the indexed assignments

$$ia_1 = 0\colon a_\bot[t \mapsto 2] \mathbin{⨟} 1\colon a_1 \text{ and } ia_2 = 1\colon a_2.$$

Their composition $ia_1 \otimes ia_2$ is the indexed assignment

$$ia_1 \otimes ia_2 = 0\colon a_\bot[t \mapsto 2] \mathbin{⨟} 1\colon \underbrace{a_\bot[p \mapsto p+1][m \mapsto t]}_{a_1 \otimes a_2}$$

Definition 23 (Composition of Symbolic Probability Distributions)**.** Let Var_1 and Var_2 be sets of typed variables, $PV_1 \subseteq \mathit{Var}_1$, $PV_2 \subseteq \mathit{Var}_2$ and $\mathit{Loc}_1, \mathit{Loc}_2$ be two finite sets of locations.

For two symbolic probability distributions $D_1 \in \mathit{SDist}(\mathit{Var}_1, \mathit{Loc}_1)$ and $D_2 \in \mathit{SDist}(\mathit{Var}_2, \mathit{Loc}_2)$, their composition $D_1 \otimes D_2 \in \mathit{SDist}(\mathit{Var}_1 \cup \mathit{Var}_2, \mathit{Loc}_1 \times \mathit{Loc}_2)$ is given by

$$(D_1 \otimes D_2)(ia, \langle \ell_1, \ell_2 \rangle) = \sum_{\substack{ia_1 \in \mathit{IAsg}(\mathit{Var}_1) \\ ia_2 \in \mathit{IAsg}(\mathit{Var}_2) \\ ia_1 \otimes ia_2 = ia}} D_1(ia_1, \ell_1) \cdot D_2(ia_2, \ell_2)$$

Composing two symbolic probability distributions composes the individual indexed assignments and sums over such elements ia_1 and ia_2 that result in the same (composed) indexed assignment ia. This is necessary, because the composition of two different pairs of indexed assignments may result in the same composite indexed assignment.

Example 12. Reconsider the automata $\mathfrak{A}_S$ and $\mathfrak{A}_R$ from Example 9 and Example 10 and the (indexed) assignments ia_1, ia_2 from Example 11. Let a_3 be the assignment $a_\perp[t \mapsto 1]$ and ia_3 be the indexed assignment $ia_\perp[0 \mapsto a_3]$. Consider the symbolic probability distributions D_1 and D_2 with

$$\begin{aligned} D_1(ia_1, wait) &= 1/3 \qquad & D_2(ia_2, busy) &= 9/10 \\ D_1(ia_3, wait) &= 2/3 \qquad & D_2(ia_\perp, idle) &= 1/10 \end{aligned}$$

and all other values mapped to 0. Their composition $D = D_1 \otimes D_2$ is given by

$$D(ia, \langle \ell_1, \ell_2 \rangle) = \begin{cases} 3/10 & \text{if } \ell_1 = wait \wedge \ell_2 = busy \wedge ia = ia_1 \otimes ia_2 \\ 1/30 & \text{if } \ell_1 = wait \wedge \ell_2 = idle \wedge ia = ia_1 \\ 3/5 & \text{if } \ell_1 = wait \wedge \ell_2 = busy \wedge ia = ia_2[0 \mapsto a_3] \\ 1/15 & \text{if } \ell_1 = wait \wedge \ell_2 = idle \wedge ia = ia_3 \end{cases}$$

Since the composition of the indexed assignments do not coincide for any pair in this case, the composition amounts to a pairwise multiplication of probabilities and composition of assignments.

Now, we have all ingredients to formally define the semantics of an NSMA in terms of a single symbolic Markov automaton.

Definition 24 (Semantics of an NSMA)**.** Let $\mathfrak{N} = \langle \mathfrak{A}_1, \ldots, \mathfrak{A}_n, Syn \rangle$ be an NSMA with symbolic Markov automata

$$\mathfrak{A}_i = \left\langle Loc_i, Var_i = PV_i \uplus TV_i, \nu_{TV,i}, TL_i, Act_i, \ell_i^0, Init_i^0, E_i \right\rangle .$$

The semantics of $\mathfrak{N}$ is the symbolic Markov automaton

$$[\![\mathfrak{N}]\!] = \left\langle Loc, Var = PV \uplus TV, \nu_{TV}, TL, Act, \ell^0, Init^0, E \right\rangle$$

where

» $Loc = \times_{i=1}^{n} Loc_i$

» $PV = \bigcup_{i=1}^{n} PV_i$,

» $TV = \bigcup_{i=1}^{n} TV_i$,

» $\nu_{TV} = \bigoplus_{i=1}^{n} \nu_{TV,i}$,

» $TL(\ell_1, \dots, \ell_n) = \bigotimes_{i=1}^{n} TL(\ell_i)$,

» $Act = \bigcup_{i=1}^{n} Act_i$,

» $\ell^0 = \langle \ell_1^0, \dots, \ell_n^0 \rangle$

» $Init^0 = \bigwedge_{i=1}^{n} Init_i^0$

» E is the unique smallest relation such that the inference rules (Indep), (Markov) and (Sync) in Figure 3.5 hold, where $D_{\ell_j} \in SDist(Var_j, Loc_j)$ is given by

$$D_{\ell_j}(ia, \ell) = \begin{cases} 1 & \text{if } ia = ia_\perp \land \ell = \ell_j \\ 0 & \text{otherwise.} \end{cases}$$

Composing a network $\mathfrak{N}$ of symbolic Markov automata yields a single symbolic Markov automaton $\mathfrak{A} = [\![\mathfrak{N}]\!]$ that captures the behavior of the whole network. To this end, $\mathfrak{A}$'s location set is the Cartesian product of the location sets of the network's SMAs. The behavior (via edges) in $[\![\mathfrak{N}]\!]$ then stems from three different sources. First, one of the automata of the network may be able to perform an unobservable step, i. e. possess an edge that is labeled with the silent action τ that is enabled in the current location, which is reflected by rule (Indep). Second, the rule (Markov) states that the automata may independently take Markovian edges. As multiple Markovian edges from different automata may have the same guard, the same rate (expression) and give rise to the same composite distribution, special attention needs to be paid in this case. We therefore sum the rates of edges that agree on the guard and the composite distribution. And finally, the rule (Sync) adds composite edges resulting from the individual synchronization vectors. In the latter case, the composite edge may only be taken if the guards of *all* participating edges are satisfied. The composition of symbolic probability distributions ensures both that (i) all "active" automata change their locations and update their private (or global) variables, and (ii) all other automata remain in their location and keep the values of their variables. Note that the premise of the (Sync) rule states that there need to be

$$\frac{\langle g, \tau, D\rangle \in E_i(\ell_i)}{\left\langle g, \tau, \bigotimes_{j=1}^{n} \begin{cases} D & \text{if } i = j \\ D_{\ell_j} & \text{otherwise} \end{cases} \right\rangle \in E(\ell_1, \ldots, \ell_n)} \ (\text{Indep})$$

$$\frac{E_g^D = \bigcup_{i=1}^{n} \left\{ e^M \in E^M_{\mathfrak{A}_i}(\ell_i) \mid g(e^M) = g \wedge D = \bigotimes_{j=1}^{n} \begin{cases} D & \text{if } i = j \\ D_{\ell_j} & \text{otherwise} \end{cases} \right\} \neq \varnothing}{\left\langle g, \sum_{e^M \in E_g^D} \lambda(e^M), D \right\rangle \in E(\ell_1, \ldots, \ell_n)} \ (\text{Markov})$$

$$\frac{\langle \alpha_1, \ldots, \alpha_n, \alpha\rangle \in Syn \wedge \forall i\, (\alpha_i \in Act_i \implies \langle \alpha_i, g_i, D_i\rangle \in E_i(\ell_i))}{\left\langle \bigwedge_{\substack{1 \le i \le n \\ \alpha_i \in Act_i}} g_i, \alpha, \bigotimes_{j=1}^{n} \begin{cases} D_j & \text{if } \alpha_j \in Act_j \\ D_{\ell_j} & \text{otherwise} \end{cases} \right\rangle \in E(\ell_1, \ldots, \ell_n)} \ (\text{Sync})$$

Figure 3.5: The inference rules for the transitions of $[\![\mathfrak{N}]\!]$.

participating edges from all automata $\mathfrak{A}_i$ for which the entry α_i in the synchronization vector is not $\perp$, which requires the automaton $\mathfrak{A}_i$ to join the synchronization.

Example 13. Reconsider the NSMA $\mathfrak{N}$ from Example 10. Figure 3.6 depicts the SMA $[\![\mathfrak{N}]\!]$. Note that the composed system synchronously performs the edges labeled with

» *send* (from $\mathfrak{A}_S$) and *reject* (from $\mathfrak{A}_R$), and

» *send* (from $\mathfrak{A}_S$) and *accept* (from $\mathfrak{A}_R$).

In contrast to that, along its Markovian edges no synchronization occurs. In particular, the location $\langle send, idle\rangle$ has a τ-labeled edge e_2^P whose symbolic probability distribution is given by D from Example 12.

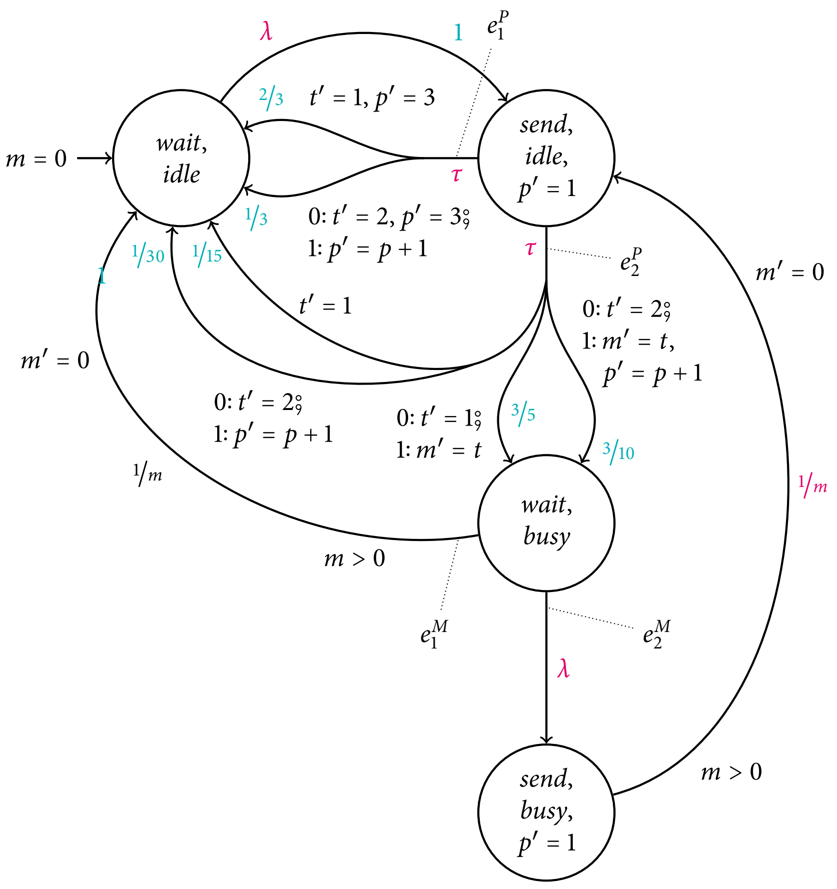


Figure 3.6: The semantics of the NSMA $\mathfrak{N}$ of Example 10.

3.3.3 Operational Semantics of SMA

Finally, we turn to the concrete semantics of a symbolic Markov automaton. For the remainder of this section, we let

$$\mathfrak{A} = \left\langle Loc, Var = PV \uplus TV, v_{TV}, TL, Act, \ell^0, Init^0, E \right\rangle$$

be a symbolic Markov automaton and $S = Loc \times Val(PV)$. Furthermore, for a variable valuation $v \in Val(PV)$, we use $v_t = v \oplus v_{TV}$ to denote the variable valuation that extends v with the default values of the transient variables.

Definition 25 (Semantics of Symbolic Probability Distributions)**.** The semantics of a symbolic probability distribution $D \in SDist(Var, Loc)$ in $\mathfrak{A}$ is the probability distribution

$$\begin{aligned} &[\![D]\!]_{\mathfrak{A}} \colon Val(Var) \to Dist(S) \\ &[\![D]\!]_{\mathfrak{A}}(v)(\ell', v') = \sum_{\substack{ia \in IAsg(Var) \\ [\![ia]\!](v_t)|_{PV} = v'}} D(ia, \ell')@v_t \end{aligned}$$

The semantics of a symbolic probability distribution can be described as follows. Given a variable valuation v, it updates the automaton location to ℓ' and valuation v according to an indexed assignment ia and assigns to this successor state a probability that corresponds to $D(ia, \ell')$ when evaluated at v_t. We have to take into account that different assignments may map the variable valuation v to the same successor valuation v' in the sense that they agree on the variables of the *permanent* variables. As transient variables have no history, and all these assignments lead to the same state in S, we sum the probabilities of all these assignments. Furthermore, it is important to extend the variable valuation v with the default values of the transient variables v_{TV}, because these values can be read by the expressions contained in the assignments.

Example 14. Reconsider the SMA $\mathfrak{A} = [\![\mathfrak{N}]\!]$ from Example 10. Let $v \in Val(\{m\})$ given by $v(m) = 0$ and D the symbolic probability distribution as in Example 12 on page 61. Then $[\![D]\!]_{\mathfrak{A}}(v)$ is the distribution

$$[\![D]\!]_{\mathfrak{A}}(v)(\ell', v') = \begin{cases} 3/5 & \text{if } \ell' = \langle wait, busy \rangle \wedge v'(m) = 1 \\ 3/10 & \text{if } \ell' = \langle wait, busy \rangle \wedge v'(m) = 2 \\ 1/10 & \text{if } \ell' = \langle wait, idle \rangle \wedge v'(m) = 0 \end{cases}$$

The first two cases represent the probabilities that the system processes a message 1 or 2, respectively, whereas the last case represents the probability that the message is dropped. The symbolic probability distribution distinguished four cases, yet the concrete distribution only three cases. This stems from the fact that the symbolic distribution maintained the assignments to the transient variables. However, as the definition of $[\![D]\!]_{\mathfrak{A}}(v)$ sums the probabilities of assignments that agree on the values of the permanent variables (i. e. $1/30 + 1/15 = 1/10$), the assignment to the transient variables is no longer visible.

Definition 26 (Joint Markovian Distribution)**.** Let $s = \langle \ell, v \rangle \in S$. Then

$$E^M_{\mathfrak{A}}(s) = \left\{ \langle g, \lambda, D \rangle \in E^M_{\mathfrak{A}}(\ell) \mid v \vDash g \right\}$$

is the set of Markovian edges enabled at s and

$$\lambda^M_s = \sum_{e^M_i \in E^M_{\mathfrak{A}}(s)} \lambda(e^M_i)@v_t$$

is the total rate of these edges.

Provided $\lambda^M_s > 0$, the *joint Markovian distribution* μ^M_s at state s is given by

$$\mu^M_s(s') = \frac{1}{\lambda^M_s} \sum_{e^M_i \in E^M_{\mathfrak{A}}(s)} (\lambda(e^M_i)@v_t) \cdot [\![D]\!]_{\mathfrak{A}}(v_t)$$

Example 15. Let $s = \langle \mathit{wait}, \mathit{busy}, m \mapsto 1 \rangle \in \mathit{Loc} \times \mathit{Val}(\{m\})$. The enabled Markovian edges at s are $E^M_{\mathfrak{A}}(s) = \left\{ e^M_1, e^M_2 \right\}$ (see Figure 3.6). Then, the total rate of Markovian edges enabled at s is $\lambda^M_s = \lambda + 1$. Hence, the joint Markovian distribution μ^M_s is given by

$$\mu^M_s(\ell', v') = \begin{cases} \dfrac{1}{\lambda + 1} & \text{if } \ell' = \langle \mathit{wait}, \mathit{idle} \rangle \wedge v'(m) = 0 \\ \dfrac{\lambda}{\lambda + 1} & \text{if } \ell' = \langle \mathit{send}, \mathit{busy} \rangle \wedge v'(m) = 1 \end{cases}$$

$$\frac{\langle g, \alpha, D\rangle \in E^P_{\mathfrak{A}}(\ell, v) \land \mu = [\![D]\!]_{\mathfrak{A}}(v)}{\langle \langle \ell, v\rangle, \alpha, \mu\rangle \in \Delta} \ (\textsc{Prob})$$

$$\frac{\lambda^M_s > 0 \land \mu = \mu^M_s}{\langle s, \Lambda, \mu\rangle \in \Delta} \ (\textsc{Markov})$$

Figure 3.7: The inference rules for the transitions of $[\![\mathfrak{A}]\!]$.

Definition 27 (Semantics of an SMA). Similar to $E^M_{\mathfrak{A}}(\ell, v)$, let

$$E^P_{\mathfrak{A}}(\ell, v) = \left\{\langle g, \alpha, D\rangle \in E^P_{\mathfrak{A}}(\ell) \mid v \vDash g\right\}$$

be the set of enabled probabilistic edges at $s = \langle \ell, v\rangle \in S$.

The semantics of $\mathfrak{A}$ is the Markov automaton

$$[\![\mathfrak{A}]\!] = \left\langle S, S^0, Act, \Delta, E\right\rangle$$

with

» $S^0 = \left\{\left\langle \ell^0, v\right\rangle \mid v \vDash Init^0\right\}$,

» Δ is the (unique) smallest relation satisfying the inference rules (Prob) and (Markov) in Figure 3.7, and

» $E(\ell, v) = \lambda^M_s$.

The state space of the resulting Markov automaton $\mathcal{M} = [\![\mathfrak{A}]\!]$ is the Cartesian product of the locations and variable valuations of the *permanent variables*. That is, only the permanent variables' values are stored in the state space. Intuitively, the values of transient variables are set to their default values at the beginning of a transition and reset to these values at the end of the transition, making it unnecessary to store them in the state. The initial states of $\mathcal{M}$ are all those states that consist of the initial location of

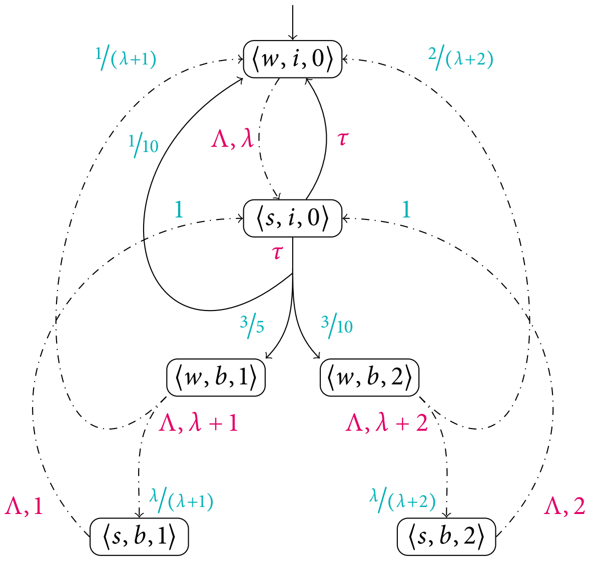


Figure 3.8: The operational semantics of the SMA $[\![\mathfrak{N}]\!]$ from Example 13.

the automaton and a variable valuation that satisfies its initial condition.

The probabilistic behavior of a state $s = \langle \ell, v \rangle$ in $\mathcal{M}$ is the union of all concrete probability distributions that are induced by the symbolic distributions of probabilistic edges enabled at s at the variable valuation v. For the Markovian behavior of s, we combine the effects of all Markovian edges enabled at s by summing the rates that lead to the same state and adding the single joint Markovian distribution at s.

Note that we defined the semantics of the SMA as an MA without state labeling. In practice, along with the SMA, we have a set of properties we are interested in that uses a finite set $AP \subseteq Bxp(Var)$ of Boolean expressions over the variables of the automaton. We can then extend the MA $[\![\mathfrak{A}]\!]$ with a labeling L by setting

$$L(\ell, v) = \{ b \in AP \mid v_t \vDash b \} .$$

Example 16. Recall the SMA $\mathfrak{A} = [\![\mathfrak{N}]\!]$ from Example 13. The operational semantics of $\mathfrak{A}$ in terms of a Markov automaton $\mathcal{M} = [\![\mathfrak{A}]\!] = \langle S, S^0, Act, \Delta, E \rangle$ is depicted in Figure 3.8 where the states are represented as tuples $\langle \ell_1, \ell_2, m \rangle$ whose last components carry the value of the (only permanent) variable m and the locations ℓ_1 and ℓ_2 of the automata $\mathfrak{A}_S$ and $\mathfrak{A}_R$ are abbreviated by their first character for conciseness.

Reconsider the symbolic probability distribution D from Example 12 and the concrete probability distribution $\mu = [\![D]\!]_{\mathfrak{A}}(\nu)$ from Example 14 for the variable valuation $\nu \in Val(\{m\})$ with $\nu(m) = 0$. As $e_2^P = \langle true, \tau, D \rangle \in E(\langle send, idle \rangle)$ (see Example 13), we have $\langle \langle send, idle, 0 \rangle, \tau, \mu \rangle \in \Delta$. μ summarizes two arcs of D, namely the ones with the indexed assignments ia_1 and ia_3 from Example 11 and Example 12 leading to location $\langle wait, idle \rangle$. Intuitively, this is because these indexed assignments only assign to transient variables and this change is not visible in the state space that only stores the values of permanent variables.

Recall the joint Markovian distribution μ_s^M at state $s = \langle wait, busy, 1 \rangle$ from Example 15. We have $\langle s, \Lambda, \mu_s^M \rangle \in \Delta$ and $E(s) = \lambda_s^M$.

3.3.4 Modeling Errors

It is apparent that the semantics of an SMA $\mathfrak{A}$ is not well-defined under all circumstances. Here, we want to mention scenarios for which $[\![\mathfrak{A}]\!]$ is not a Markov automaton and that are therefore regarded as a modeling error.

First, the symbolic probability distributions may induce a function that is not a probability distribution. For example, it may be the case that the expressions defining the probabilities evaluate to negative values or do not sum to one. Second, the rate expressions of Markovian edges can potentially evaluate to a nonpositive value.

Third, composing edges during parallel composition may result in another form of modeling error. To see this, recall that our definition of the composition $a_1 \otimes a_2$ of two assignments (see Definition 22) maps a variable x to $\top$ if both a_1 and a_2 assign a value to it, as it would be unclear which of the values is to be preferred. In the context of composing edges, assignments are composed to reflect that along the composed edges all of these updates to variables are to be executed. If two assignments that are composed write to the same variable x, the resulting assignment will map x to $\top$ to indicate the ill-definedness. Similarly, when composing the transient location assignments of a network, shared transient variables may illegally be written multiple times.

Finally, the parallel composition of NSMA may produce an ill-defined SMA if several automata of the network use shared transient variables with deviating default values.

Except for the last modeling error, **JANI** chooses to ignore this ill-definedness as long as it does not appear at any state (or variable valuation) that is reachable from the initial states of $[\![\mathfrak{A}]\!]$. After all, this illegal behavior does not affect the observable behavior of the system.

3.3.5 Reward Models

To enable the specification of other measures of interest, an SMA $\mathfrak{A}$ can be equipped with a symbolic reward model.

Definition 28 (Symbolic Reward Model)**.** A symbolic reward model (SRM) *rew* for an SMA $\mathfrak{A} = \langle Loc, Var = PV \uplus TV, v_{TV}, TL, Act, \ell^0, Init^0, E \rangle$, is an expression $rew \in Qxp(Var)$.

Intuitively, a symbolic reward model is simply an expression *rew* over the (transient and permanent) variables of the SMA $\mathfrak{A}$. The reward earned in a state $s = \langle \ell, v \rangle$ is equal to what *rew* evaluates to in s. As symbolic reward models may also involve transient variables, their default values are assumed in state s unless the transient location assignment $TL(\ell)$ assigns a different value to them.

In a similar way, rewards are associated with transitions. Intuitively, a transition from state $s = \langle \ell, v \rangle$ generated by an element $\langle ia, \ell' \rangle \in supp(D)$ yields an amount of reward equal to *rew* evaluated at the variable valuation that is obtained when applying *ia* to the variable valuation $v \oplus v_{TV}$.

Definition 29 (Semantics of an SRM)**.** Let *rew* be an SRM for the SMA $\mathfrak{A}$. The semantics of $\mathfrak{A}$ and *rew* is an MRA

$$[\![\mathfrak{A}, rew]\!] = \langle S, S^0, Act, \Delta, E, r \rangle$$

where S, S^0, Act and E are defined as in Definition 27, the state reward function is given as

$$r(\ell, v) = rew@[\![TL(\ell)]\!](v_t)$$

and the transition relation Δ (including the rewards) is the smallest relation such that rules (Prob) and (Markov) from Figure 3.9.

For $s = \langle \ell, v \rangle, s' = \langle \ell', v' \rangle$, the rules are given as follows.

$$\frac{\langle g, \alpha, D \rangle \in E^P_{\mathfrak{A}}(s) \wedge \mu = [\![D]\!]_{\mathfrak{A}}(v) \wedge \rho = rew(s, \{e^P\}, \mu)}{\langle \langle \ell, v \rangle, \alpha, \rho, \mu \rangle \in \Delta} \quad (\textsc{Prob})$$

$$\frac{\mu = \mu^M_s \wedge \rho = rew(s, E^M_{\mathfrak{A}}(s), \mu)}{\langle \langle \ell, v \rangle, \Lambda, \rho, \mu \rangle \in \Delta} \quad (\textsc{Markov})$$

where for an edge $e \in E'$ in a set of edges E' we let

$$P(e) = \begin{cases} \dfrac{\lambda(e)@v_t}{E(s)} & \text{if } e \in E^M_{\mathfrak{A}} \\ 1 & \text{otherwise} \end{cases}$$

be the relative probability of e in

$$\begin{aligned} rew(s, E', \mu)(s') &= \frac{rew(s, E')(s')}{\mu(s')} \\ rew(s, E')(s') &= \sum_{e \in E'} P(e) \cdot \sum_{\substack{ia \in IAsg(Var) \\ [\![ia]\!](v_t)|_{PV} = v'}} D(ia, \ell')@v_t \cdot rew@[\![ia]\!](v_t) \end{aligned}$$

Figure 3.9: The inference rules for the transitions of $[\![\mathfrak{A}, rew]\!]$.

The definition of the state reward function is straightforward, whereas the transition rewards are slightly more involved. Starting in a state $s = \langle \ell, v \rangle$ there may be several indexed assignments of the same probabilistic edge e^P that result in the same destination state $s' = \langle \ell', v' \rangle$ but disagree on the reward values. In the Markov automaton, they will be summarized in the form of a single transition from s to s'. In this case, the expected reward earned by an indexed assignment *ia* must be weighted with its share of the overall probability $\mu(s')$ of moving from s to s' with μ. Additionally, as in MA, multiple Markovian edges enabled in a state s are summarized in the joint Markovian distribution. Therefore, the probabilities of the assignments of the individual edges need to be additionally scaled with the ratio of the rate of the rewarded edge with the total exit rate of s.

It is considered a modeling error if $[\![\mathfrak{A}, \mathit{rew}]\!]$ is not a proper MRA, which can happen if *rew* evaluates to a negative number. As for the modeling errors in $\mathfrak{A}$, **JANI** considers this to be an error only if it happens in the reachable fragment of $[\![\mathfrak{A}]\!]$.

Example 17. Reconsider the SMA $\mathfrak{A} = [\![\mathfrak{N}]\!]$ from Example 13 and the reward model $\mathit{rew} = p$ that measures the penalty that a system run incurs.

For the most part, the MRA $\mathcal{M} = [\![\mathfrak{A}, \mathit{rew}]\!] = \langle S, S^0, \mathit{Act}, \Delta, E, r \rangle$ coincides with the SMA $[\![\mathfrak{A}]\!]$. For the state rewards, we have

$$r(\ell, v) = \begin{cases} 1 & \text{if } \ell = \langle \mathit{send}, \mathit{idle} \rangle \lor \ell = \langle \mathit{send}, \mathit{busy} \rangle \\ 0 & \text{otherwise} \end{cases}$$

because (i) *rew* only evaluates to a non-zero value in locations with a transient location assignment that assigns to p (p's default value is 0) and (ii) the locations $\langle \mathit{send}, \mathit{idle} \rangle$ and $\langle \mathit{send}, \mathit{busy} \rangle$ both assign the constant 1 to p.

Consider the state $s = \langle \mathit{send}, \mathit{idle}, 0 \rangle$ and the distribution μ from Example 14 arising from the symbolic probability distribution D from Example 11 of edge e_2^P. As argued in Example 14, it summarizes the probability of two indexed assignments ia_1 and ia_3, because they are indistinguishable on the concrete state space. However, their behavior is visible in terms of transition rewards, because assignments to transient variables influence the value of *rew*. ia_3 makes *rew* evaluate to 0, because it does not assign to p and therefore p's default value is assumed. In contrast, ia_3 yields a value of 1 with respect to *rew*. Consequently, we have

$$\langle s, \tau, \rho, \mu \rangle \in \Delta$$

where

$$\rho(s') = \begin{cases} 1/3 & \text{if } s' = \langle \mathit{wait}, \mathit{idle}, 0 \rangle \\ 1 & \text{if } s' = \langle \mathit{wait}, \mathit{busy}, 2 \rangle \\ 0 & \text{otherwise} \end{cases}$$

because

$$\mathit{rew}(s, e_1^P, \mu, \langle \mathit{wait}, \mathit{idle}, 0 \rangle) = \underbrace{\frac{1/15}{1/10}}_{\mathit{ia}_1} \cdot 0 + \underbrace{\frac{1/30}{1/10}}_{\mathit{ia}_3} \cdot 1 = 1/3$$

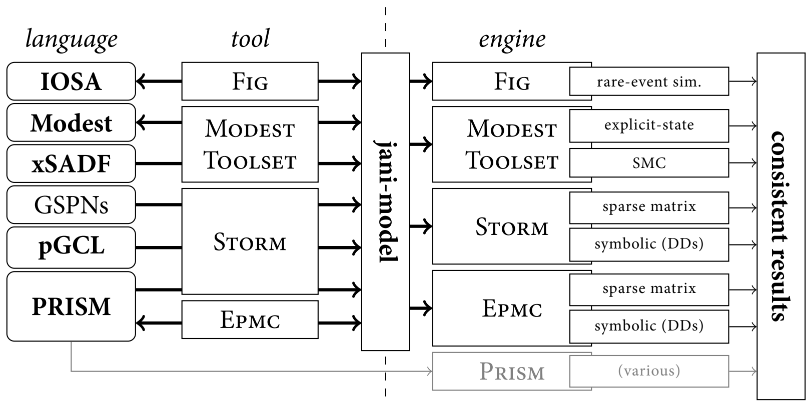


Figure 3.10: The **JANI** landscape [Bud+17].

and

$$rew(s, e_1^P, \mu, \langle wait, busy, 2\rangle) = \frac{3/10}{3/10} \cdot 1 = 1.$$

3.4 Tool Support

In a collective effort, **JANI** is supported by several major tools of the probabilistic model checking community despite being a very young language. In addition to analyzing models given in the **JANI** language, most of these tools support translating one or more input languages to **JANI** and sometimes even in the reverse direction. Figure 3.10 gives an overview of the current tool landscape. More specifically, it displays all possible input languages and formalisms and which of the involved tools are able to translate them to **JANI**or vice versa. Also, it shows the various engines (data structures and model checking approaches) that can be used to analyze **JANI** models and obtain consistent results. We will now first give a brief overview over the input languages that can be transformed to **JANI** and then proceed with details on the tools and engines.

3.4.1 Modeling Languages

While **JANI** is designed to be human-readable to allow for manual debugging, we do not expect users to directly formulate their input models in the **JANI** format. Rather, users can select from a variety of modeling languages from several domains that are then automatically translated. Note that while translations are only supported by one or two tools, all input models in any of the supported languages can in principle be used as input to any of the tools: after converting the selected model to the **JANI** format with the specific tool, all other tools are able to treat the model unless it contains features of **JANI** that are explicitly unsupported by the target tool.

In the long run, a graphical user interface that supports the direct generation of **JANI** models without intermediate translations or the need for writing **json** input would be immensely helpful. Similar to Uppaal, the user could then draw the symbolic automata and edges in a graphical editor and benefit from "what-you-see-is-what-you-get" in the heavily simplified modeling process. However, as of today, no such editor exists. We will now briefly describe the modeling languages that are currently connected to **JANI**.

PRISM. Although **PRISM** has been mentioned several times before, we want to recap its essential properties here. **PRISM** is the input language to the famous probabilistic model checker Prism. It is based on reactive modules over symbolic variables [AH99] that operate in parallel using *commands*. Every command consist of a guard that governs when it is enabled and a probability distribution over variable updates that determines the states of the system after having executed the command. This process either happens independently and only one of the modules executes a command, or in a synchronizing fashion, i. e. several modules change their internal state simultaneously. This fundamental similarity to **JANI** is not coincidental: because of the popularity of Prism and the fact that many other tools currently support extended subsets of **PRISM**, many of the ideas of **JANI** ultimately go back to the **PRISM** language.

Through both Storm and Epmc, we are able to make the large Prism benchmark suite [KNP12] available in the **JANI** format. Epmc even supports the reverse transformation and enables the treatment (of a restricted class) of **JANI** models with the large number of verification backends found within Prism.

Example 18. Figure 3.11 shows a simple probabilistic timed automaton (PTA) specified in the **PRISM** language. The PTA has two locations that are encoded by the variable `l` and one clock `c`. As long as the system is in location 1, the value of the clock is required to be at most 2. In location 0, the system tries to send a message to the channel, which succeeds with probability 0.99 and fails otherwise. After a

```
pta

module Channel
    l: [0..1] init 0; // control location
    c: clock; // measuring delay

    invariant
       l=1 ⇒ c <= 2
    endinvariant

    [send] l=0 → 0.01: (l'=0) + 0.99: (l'=1)&(c'=0);
    [receive] l=1 & c >= 2 → 1: (l'=0);
endmodule
```

Figure 3.11: A channel PTA modeled in the **PRISM** language.

successful send, the automaton starts the clock c to measure the delay until it receives an acknowledgement that the message arrived properly. Due to the invariant, the automaton will re-send the message after exactly two time units.

IOSA. In stochastic automata (SA) [DK05] arbitrarily (continuously) distributed clocks govern the occurrence of events in the system. Input-Output stochastic automata (I/O SA) [DLM16] are networks of stochastic automata that syntactically guarantee the absence of nondeterminism. To this end, they require (i) automata to be input-enabled (they must be able to respond to an action at all times), (ii) that any event can only be emitted by at most one automaton, and (iii) that clocks can only control the timing of outputs. **IOSA** enables the specification of I/O SA in the form of a language that is a slight variation of **PRISM**. Its synchronization mechanism uses input and output events whose names end in a "?" and "!", respectively. The communication between automata is limited to that of *broadcasting*. That is, an output event needs to synchronize with all matching input events. All automata that do currently not offer synchronizing with the output event do not prevent progress as they are input-enabled.

The Fig tool [BDM16] translates **IOSA** to and from **JANI**. When converting **IOSA** to **JANI**, the resulting model type is a stochastic timed automaton (STA) as I/O SA are a proper subset of STA. In the reverse direction, input STA and CTMCs in the **JANI** format need to specify synchronization vectors that correspond to broadcast communication in order to be supported.

```
const int c = 10;
const int λ = 3;
const int μ = 2;

module Arrivals
  arrival: clock;  // external arrivals ~ Exponential(λ)
  [arrive!] @ arrival → (arrival'=exponential(λ));
endmodule

module Queue
  q: [0..c];
  process: clock;  // queue processing ~ Exponential(μ)
  // Packet arrival
  [arrive?] q=0 → (q'= q+1)&(process'=exponential(μ));
  [arrive?] q>0 & q<c → (q'=q+1);
  [arrive?] q=c → (q'=c);
  // Packet processing
  [send!] q=1 @ process → (q'=q-1);
  [send!] q>1 @ process → (q'=q-1)&(process'=exponential(μ));
endmodule
```

Figure 3.12: A simple queue modeled in **IOSA**.

Example 19. Figure 3.12 shows an **IOSA** example. It models a (bounded) queue that is processing packages that arrive with an exponential distribution with mean λ. When the queue is non-empty, packages are processed and sent, which takes an amount of time that is also exponentially distributed with parameter μ. To ensure that arriving packages from the module `Arrivals` are correctly registered (with respect to the bounded size of the queue), the module `Queue` provides appropriate input events.

Modest. The **Modest** language is a high-level modeling language that supports very expressive features like recursive data-structures and process invocations, loops and exception handling. Its semantics are defined in terms of STA [Boh+06] and were later extended to SHA [Hah+13]. As SHA cover all model types that are currently targeted by **JANI** (see Figure 3.1), **Modest** is the richest modeling language that is currently connected to **JANI**. As indicated in Figure 3.10, the MODEST TOOLSET [HH14] allows for translating the language to **JANI** and vice versa.

```
process Channel() {
  send palt {
    :99: delay(2)
      receive
    : 1: // message lost
      {==}
  };
  Channel();
}
Channel();
```

Figure 3.13: The channel model in the **Modest** language.

```
int c := 0;
int x := 0;

while (c = 0) {
  {
    x := x + 1
  } [0.5] {
    c := 1
  }
}
observe (x % 2 = 1);
```

Figure 3.14: A example program in **pGCL**.

Example 20. Reconsider the model of a lossy channel from Example 18. Figure 3.13 shows the same model in the **Modest** syntax.

pGCL. Probabilistic programming languages extend standard languages with constructs to sample from random distributions. An example for such a language is the probabilistic guarded command language (**pGCL**) [MM05] with observe statements [Kat+15]. It additionally provides a mechanism to *condition* the probability distributions that are described by such programs. These constructs are at the heart of algorithms in machine learning, security, and quantum computing [Gor+14]. The operational semantics of a probabilistic program written in **pGCL** is a (typically infinite) MDP. Storm implements a translation from **pGCL** to **jani-model** via program graphs.

Example 21. An example program in the **pGCL** language is shown in Figure 3.14. In each iteration of the while-loop, a fair coin is thrown and depending on the outcome x is either increased or the iteration aborted. Overall, the value of x at the end of the while loop is geometrically distributed with success probability one half. The observe statement at the end of the program filters all possible program runs that reach this point with a value of x. Stated differently, only runs that assign an odd value to x have a non-zero probability of terminating. To make up for the "lost" probability mass, the probability of the remaining events is rescaled, which formally amounts to probabilistic conditioning. Intuitively, this results in x being geometrically distributed over the odd numbers only.

A noticeable feature of the translation of **pGCL** to **JANI** in Storm is the detection of transient variables. It uses heuristics to determine whether the history of a variable is relevant or not within a **pGCL** program.

Under some restrictions the value of a variable does not need to be stored explicitly and the variable can be treated as transient. Avoiding to store an unbounded variable may make the underlying PA finite and therefore amenable to standard probabilistic model checking. Consider as an example a modified version of the program in Example 21 that omits the observe statement at the end. If the property in question is the expected value of x, the increments to this variable can be seen as obtaining a reward of 1. The question can then be answered by an expected reward analysis on a finite PA.

GSPNs. Petri Nets (PN) are among the most widespread modeling formalisms for concurrent processes. Generalised stochastic Petri nets (GSPN) [MCB84] extend the standard *immediate* transitions of PNs with *exponentially delayed* ones. Most often, the semantics of GSPN were formally specified in terms of a CTMC. However, this requires resolving the nondeterminism from immediate transitions by specifying weights in the GSPN. This avoids nondeterminism by implicitly encoding a probabilistic branching, which may be unnatural depending on the system's domain and environment. [Eis+13] presents a formal semantics for *every* GSPN, including *confused* ones with actual nondeterminism, in terms of Markov automata.

Based on an implementation of this semantics, Storm can translate GSPNs given either as a GreatSPN project [Amp14] or in a variant of the ISO-standard **PNML** [ISO11b] format into a **jani-model** description. The encoding uses variables to represent markings and probabilistic and Markovian edges for immediate and timed transitions, respectively. The translation of the (optional) weights is more involved.

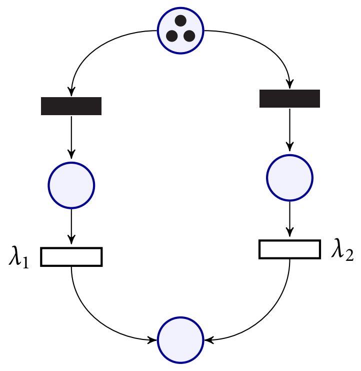


Figure 3.15: A conflicted GSPN.

Example 22. We show an example GSPN in Figure 3.15. The net has four places (circles) and four transitions (rectangles). Two transitions are immediate (black) and two are timed (white). Each of the timed transitions is equipped with a rate that is the parameter of the negative exponential distribution governing the firing times. This net is what the literature refers to as *conflicted* as the current marking (the amount of tokens in each place) enables both immediate transitions and there is no policy on which to schedule first. Hence, the system needs to nondeterministically select the firing order, which is not supported when using the CTMC semantics of GSPNs. Using the Markov automaton semantics, Storm can translate every GSPN to a symbolic Markov automaton in the **JANI** format.

xSADF. Dataflow formalisms are popular in the study of embedded data processing applications. The recently introduced extensions of scenario-aware dataflow [The+06] **xSADF** [HHB16] and **eSADF** [KW16] add cost annotations (to model, for example, power consumption), nondeterminism, and continuous stochastic execution times. For **xSADF** the compositional semantics in terms of STA are implemented in the Modest Toolset. Via the latter's support for **jani-model**, we can now also convert xSADF specifications to **jani-model**. The resulting models are networks of STA that make use of some of the features specified in the `datatypes` and `functions` extensions of **JANI** to encode the unbounded typed scenario channels of **xSADF**.

3.4.2 Analysis

Using intermediate representations within tools has the well-known advantage that the same backend can be used for several input languages. This not only reduces programming and maintenance effort but also allows a more optimized and streamlined backend. A very popular example for this approach is the compiler framework Llvm with its language **LLVM-IR** [LA04].

Currently, there are four tools that support the analysis of **JANI** models directly: Fig [BDM16], Epmc [Hah+14], the Modest Toolset [HH14] and Storm [Deh+17]. However, through Epmc's capability to transform **JANI** to **PRISM**, Prism's [KNP11] broad palette of verification approaches can also be implicitly leveraged. Table 3.1 summarizes which model types, features and property types are supported by which of the tools. The columns indicate support for (i) probabilistic reachability (P), (ii) probabilistic computation tree logic (PCTL), (iii) satisfaction probabilities for linear temporal logic formulas (LTL), (iv) expected values or rewards (E), and (v) steady-state measures (S).

We refrain from describing the tools in more detail as this is the topic of Chapter 7.

3.5 Related Work

Several efforts have been made to standardize modeling languages for broader use, or to develop overarching formalisms that offer a union of the features of many different specialized languages. Notable examples for this include the ISO standard Lotos [BB87] and **CIF** language and format [ABR13]. The latter is a complex specification consisting of a textual and graphical syntax for human use plus an XML representation that covers quantitative aspects such as timed and hybrid, but not probabilistic, behavior. It has connections to a variety of tools including those based on **Modelica** [Fri11], which itself is also an open specification intended to be supported by tools focusing on continuous system and controller simulation. Effectively the only implementation of Lotos is in the Cadp toolset [Gar+13], and active **CIF** support appears restricted to the CIF 3 tool [Bee+14]. We took inspiration from the use of synchronization vectors in Cadp and related tools to compactly-yet-flexibly specify how automata interact. **AADL** is a language originating from the automotive industry and currently used in other domains as well. Together with its error annex, the resulting semantic model is a Markov automaton. The semantic model of the **PRISM** language — networks of Markov chains, probabilistic automata or probabilistic timed automata with variables — forms the conceptual basis of **jani-model**. The notion of transient variables in SMA was adopted from RDDL [San10]. The HOA format [Bab+15] is a tool-independent exchange format for ω-automata designed to represent linear-time properties for or during model checking.

Table 3.1: Support for model types, features and property classes in analysis tools.

tool	engine	LTS	DTMC	CTMC	MDP	CTMDP	MA	TA	PTA	STA	SHA	arrays	datatypes	functions	P	PCTL	LTL	E	S
Fig	rare	–	*	✓	–	–	–	–	–	(1)	–	(2)	–	*	✓	–	–	*	✓
Epmc	sparse	✓	✓	✓	✓	*	*	–	–	–	–	–	–	–	✓	✓	✓	*	*
	dd	✓	✓	✓	✓	*	*	–	–	–	–	–	–	–	✓	✓	✓	*	*
Modest Toolset	explicit	✓	✓	✓	✓	–	✓	✓	✓	✓	(3)	✓	✓	✓	✓	*	–	✓	–
	SMC	✓	✓	–	✓	–	(4)	(4)	(4)	(4)	–	✓	✓	✓	✓	–	–	✓	–
Storm	sparse	✓	✓	✓	✓	–	✓	–	–	–	–	✓	–	–	✓	✓	–	✓	✓
	dd	✓	✓	✓	✓	–	✓	–	–	–	–	(2)	–	–	✓	✓	–	✓	*
Prism	(various)	✓	✓	✓	✓	–	–	✓	✓	–	–	*	*	*	✓	✓	✓	✓	✓

✓	≙ supported
*	≙ support planned
(1)	≙ only broadcast-based input/output STA supported
(2)	≙ support planned for fixed-size arrays
(3)	≙ supported via the prohver tool
(4)	≙ supported for deterministic models

Atlantif [SLG09] is an intermediate model for real-time systems with data that can be translated to timed automata or Petri nets. In the area of satisfiability-modulo-theories (SMT) solvers, the SMT-LIB standard [BFT15] defines a widely-used data format and tool interface protocol analogous to the pair of **jani-model/jani-interaction** that we propose for quantitative verification. The formats mentioned so far provide concise high-level descriptions of potentially vast state spaces. An alternative is to exchange low-level representations of actual state spaces, representing all the concrete states of the semantics of some high-level model. Examples of such state space-level encodings include Cadp's BCG format and Mrmc's [Kat+11] input format. Disadvantages are that the file size explodes with the state space, and all structural information necessary for symbolic (e. g. MTBDD-based) reasoning or static analysis is lost.

A number of tools take a reversed approach by providing an interface to plug in different input languages. In the non-quantitative setting, one example is LTSMin [Kan+15] and its PINS interface. However, this is a C/C++ API on the state space level, so every input language needs to provide a complete implementation of its semantics for this tool-specific interface. A prominent tool with a similar approach that uses quantitative models is Möbius [Cou+09]. Notably, a command-line interface has recently been added to Möbius' existing graphical and low-level interfaces to improve interoperability [KS13]. Similarly, Prism features a **Java**-level model generator interface. The Modest Toolset [HH14] also used an internal semantic model similar to that of **jani-model** that allows it to translate and connect to various external tools, albeit via their command-line interfaces.

The **JANI** specification can be seen as a *metamodel*. The Eclipse EMF/Ecore platform [Ecl] is popular for building and working with metamodels. We chose to create a standalone specification instead in order to avoid the heavy dependency on Eclipse and to not force a preferred programming language on implementers.

Chapter **4**

Fast Debugging of JANI Models

4.1 Motivation and Goals

Model checking is a push-button technique to verify or refute the compliance of a system with a specification. It is increasingly used in industry and has had remarkable successes [BLR11; Hol14]. Two of the major challenges that remain to continue this success story are (i) improving its scalability to make it applicable to real-world problems whose complexity is ever growing, (ii) gain acceptance among engineers. For the latter, it is not enough to merely (dis)prove statement about the system, but requires *an explanation* of the verification result. In case the verification attempt fails, counterexamples have emerged as a way to capture the faulty behavior of the system. Experience has shown that counterexamples are the single most effective feature to convince system engineers of the value of formal verification [CV03]. Consequently, the competition on software verification (SV-COMP) requires the participating tools to generate counterexamples for almost all categories.

Interestingly, counterexamples also play a key role in tackling the first challenge mentioned above. State-of-the-art techniques like counterexample-guided abstraction refinement [Cla+00] (CEGAR) and counterexample-guided inductive synthesis [Sol+06] (CEGIS) use counterexamples to guide the solution process.

These observations sparked a whole body of research and made algorithmic generation of counterexamples a key component in modern model checkers such as Spin [Hol97], NuSMV [Cim+02], CPAChecker [BK11], and Ultimate Buchi Automizer [HHP13]. These tools include powerful facilities to generate counterexamples in various formats. Such counterexamples are typically provided at the modeling level, like a diagram indicat-

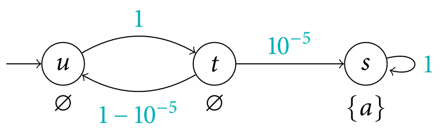


Figure 4.1: A simple DTMC [HKD09].

ing how the change of model variables yields a property violation, or a message sequence chart illustrating the failing scenario. Substantial efforts have been made to generate succinct counterexamples, often at the price of an increased time complexity [GM07; HG08; SB05].

Despite the growing popularity of probabilistic model checkers, such facilities are absent in tools such as PRISM [KNP11] and MRMC [Kat+11]. One of the reasons is that counterexamples in the probabilistic setting turn out to be more complicated. Consider, for example, a labeled transition system $\mathcal{L}$ and a distinguished "bad" state s. A safety property might now state that s is not reachable in $\mathcal{L}$. If this property does not hold, the simplest form a counterexample can take is a (finite) path in $\mathcal{L}$ that starts in the initial state and ends in s. The corresponding problem in the probabilistic setting is whether in a probabilistic automaton $\mathcal{M}$, the *probability* to reach s is below a safety threshold λ.

As an example, consider the DTMC $\mathcal{D}$ depicted in Figure 4.1 along with the probabilistic safety property $\varphi = \mathsf{P}_{\leq 9/10}(\lozenge a)$ that expresses that the probability to reach states labeled with a is at most 0.9. Clearly, $\mathcal{D} \not\models \varphi$ as the probability to reach state s is 1. However, no single path in $\mathcal{M}$ alone carries enough probability mass to violate the bound λ. Instead, a set of finite paths is required whose *combined* probability mass exceeds the threshold. Computing minimal counterexamples in terms of sets of paths amounts to solving a k-shortest path problem [HKD09]. However, for $\mathcal{D}$ and φ, more than 230000 paths are needed to explain the violation [HKD09]. In fact, it can be shown that in general exponentially many paths are necessary and for formulae with strict probability bounds it may even be the case that no finite set of paths is sufficient. Subsequent research tried to mitigate these deficiencies. While [AL10] employs heuristics to guide the exploration of the system, [WBB09] uses a symbolic encoding of the system to enable bounded model checking. [LL13] enhances the path search by post-processing steps, such as building a fault-tree to better explain the causality in the model. However, these approaches suffer from a potentially large number of paths that need to be explored. A viable alternative is to determine minimal critical subsystems [Wim+12; Wim+14], i. e., model fragments for which the likelihood of reaching bad states already exceeds the probability bound

λ. Using binary decision diagrams, minimal critical subsystems can be computed for models with billions of states [Jan+14]. Yet, the generated counterexamples remain huge. The fundamental commonality of these approaches is that they work at the state space level. The sheer size of such counterexamples makes them incomprehensible and not effectively usable in CEGAR approaches for probabilistic systems [HWZ08; CCD14].

Recent works take a radically different approach. Drawing inspiration from the efforts in the qualitative setting to lift counterexamples from the state space level to the specification level, the authors propose to map the core argument of the violation back to the (symbolic) model specification. In [Brá+15], the authors use learning techniques to identify small schedulers that represent critical decisions in the model and in [Fen+18] explanations in terms of structured language are synthesized. We, however, focus on the approach of [Wim+15]. Here, the authors determine the smallest set of commands C of a **PRISM** model [KNP11] whose semantics in terms of a PA already violate a reachability property. The restriction of the **PRISM** model to the command set C then forms a *high-level counterexample*. Algorithmically, C is computed by encoding the query in terms of an mixed-integer linear program (MILP) and delegating the solution to an off-the-shelf solver like Gurobi. Since deciding whether there is a critical set of commands of size at most k is NP-hard [Wim+15], this encoding is optimal in the sense that it does not increase the theoretical complexity. However, MILP solvers target general purpose problems and do not exploit domain-specific knowledge unless it is part of the encoding. In fact, [Wim+15] shows that the solver's performance varies drastically depending on which additional information is encoded into the problem.

In this chapter, we present a more scalable algorithm that solves the same problem whose fundamental advantage lies in leveraging the domain knowledge through the use of a probabilistic model checker. While our presentation will focus on computing high-level counterexamples for **JANI** models, the approach is applicable to other modeling formalisms for probabilistic automata such as **PRISM**, **PIOA** [Can+08], stochastic process algebras [Kat+12] and the graphical component-wise representation of systems as possible in **Uppaal** [Beh+06].

Formally, we will consider the problem of computing a minimal critical edge set.

Definition 30 (Minimal Critical Edge Set)**.** Let $\mathfrak{N} = \langle \mathfrak{A}_1, \ldots, \mathfrak{A}_n, Syn \rangle$ be a network of symbolic *probabilistic* automata

$$\mathfrak{A}_i = \left\langle Loc_i, Var_i = PV_i \uplus TV_i, \nu_{TV,i}, TL_i, Act_i, \ell_i^0, Init_i^0, E_i \right\rangle$$

and $\mathcal{M} = [\![\mathfrak{N}]\!]$ its semantics in terms of a probabilistic automaton. A set

$$E^* \subseteq \bigcup_{i=1}^{n} (Loc_i \times Bxp(Var(\mathfrak{N})) \times Act(\mathfrak{N}) \times SDist(Var(\mathfrak{N}), Loc_i))$$

of (location-extended) edges is called a *critical edge set* for $\mathfrak{N}$ and the probabilistic safety property $\varphi = \mathsf{P}_{\leq\lambda}(\lozenge T)$, if

$$[\![\mathfrak{N}|_{E^*}]\!] \not\models \varphi$$

where

» $\mathfrak{N}|_{E^*} = \left\langle \mathfrak{A}_1|_{E^*}, \ldots, \mathfrak{A}_n|_{E^*}, Syn \right\rangle$, and

» $\mathfrak{A}_i|_{E^*} = \left\langle Loc_i, Var_i = PV_i \uplus TV_i, \nu_{TV,i}, TL, Act_i, E_i \cap E^*, \ell_i^0, Init_i^0 \right\rangle$.

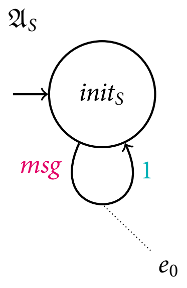
and

$$\langle \ell, g, \alpha, D \rangle \in E_i \cap E^* \iff \langle g, \alpha, D \rangle \in E(\ell) \wedge \langle \ell, g, \alpha, D \rangle \in E^*.$$

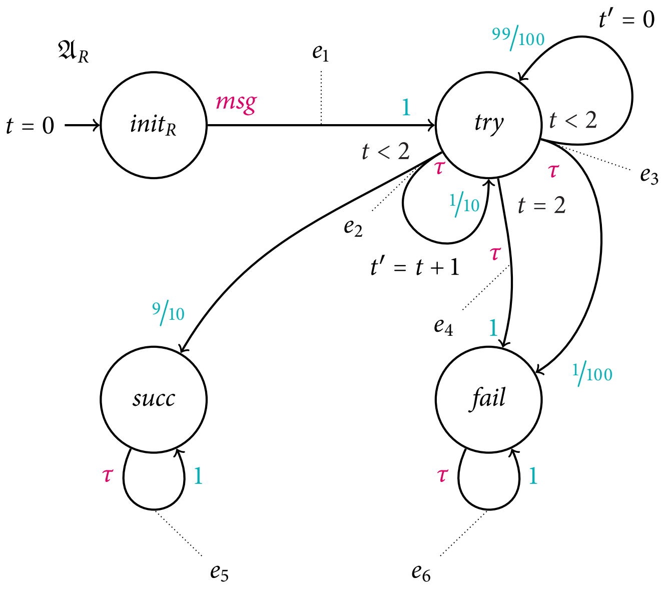
E^* is called *minimal* if there is no other critical edge set E' with $|E'| < |E^*|$.

Intuitively, a minimal critical edge set E^* for a NSPA $\mathfrak{N}$ and property φ, is a set of edges such that deleting all edges that are not in E^* from automata contained in the network will still induce a probabilistic automaton that violates the property φ. Hence, any extension of $\mathfrak{N}|_{E^*}$ will also violate φ, since — intuitively speaking — the resulting PA can only have more behavior and the maximal reachability probability can only increase. Note that this monotonicity property holds for probabilistic safety properties in particular but not for all properties in general. We remark that a minimal critical edge set E^* is not unique and there may be several such sets with size $|E^*|$.

Example 23. Consider the two symbolic probabilistic automata in Figure 4.2 that form a network of symbolic probabilistic automata $\mathfrak{N}$ together with the synchronization vectors $Syn = \{\langle msg, msg, msg \rangle\}$. $\mathfrak{A}_S$ (top) generates a simple message that $\mathfrak{A}_R$ (bottom) receives. The latter then tries at most two times to process the message. Every attempt succeeds with probability $9/10$ and fails with $1/10$. Additionally, the system has a mechanism to reset the number of remaining tries (variable t) that also causes the system to fail in 1% of the cases. The system designer uses a standard probabilistic



(a) The SPA $\mathfrak{A}_S$.



(b) The SPA $\mathfrak{A}_R$.

Figure 4.2: The two automata forming the NSPA $\mathfrak{N}$.

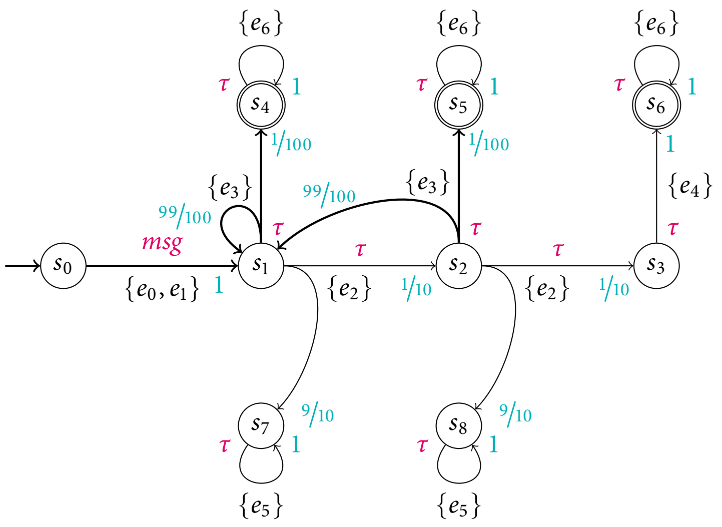


Figure 4.3: The operational semantics of $\mathfrak{N}$ in terms of a PA $\mathcal{M}$.

model checker that can handle **JANI** and asks whether $[\![\mathfrak{N}]\!] \models \varphi = \mathsf{P}_{\leq\frac{1}{5}}(\Diamond T)$ where $T \hat{=} \mathit{fail}$ to verify that on average the overall system only fails in at most every fifth run independent of the how the nondeterminism is resolved. The verifier correctly returns that the property is not satisfied and reports that the maximal reachability probability w.r.t. T is in fact 1. To see this, consider the operational semantics $[\![\mathfrak{N}]\!]$ depicted in Figure 4.3 where the set T now corresponds to $\{s_4, s_5, s_6\}$. The problem is that the reset operation is directly available *before* the first attempt of processing it and a scheduler can therefore schedule this choice whenever it is in state s_1. Strengthening the guard of edge e_2 from $t < 2$ to $t > 0 \wedge t < 2$ would remove the choice labeled with $\{e_2\}$ from s_1 and make the resulting system satisfy φ. While the NSPA $\mathfrak{N}$ has 7 edges, three of them suffice to exceed the safety threshold $\lambda = \frac{1}{5}$ of the property: a minimal critical edge set for $\mathfrak{N}$ and φ is the set $E^* = \{e_0, e_1, e_3\}$. The restriction $\mathcal{M}|_{E^*}$ of $\mathcal{M}$ to E^*, which we define formally later, is the boldly drawn fragment in Figure 4.3.

For the remainder of this chapter, we will use the term edges to refer to location-extended edges (see Definition 20) and let $\mathfrak{N} = \langle \mathfrak{A}_1, \ldots, \mathfrak{A}_n, \mathit{Syn} \rangle$ be an NSPA with SPA

$$\mathfrak{A}_i = \left\langle \mathit{Loc}_i, \mathit{Var}_i = \mathit{PV}_i \uplus \mathit{TV}_i, \nu_{\mathit{TV},i}, \mathit{TL}_i, \mathit{Act}_i, \ell^0_i, \mathit{Init}^0_i, E_i \right\rangle,$$

$\mathcal{M} = [\![\mathfrak{N}]\!] = \langle S, S^0, Act, \Delta \rangle$ be the PA induced by $\mathfrak{N}$, and $\varphi = \mathsf{P}_{\leq\lambda}(\lozenge T)$ with a target set $T \subseteq S$ be a probabilistic safety property. Furthermore, let $\mathcal{M}|_E = [\![\mathfrak{N}|_E]\!] = \langle S, S^0, Act, \Delta|_E \rangle$ be the restricted model with respect to a given set E as in Definition 30. To guarantee the existence of a (minimal) critical edge set, we assume $\mathcal{M} \not\models \varphi$. This property is easily checked using standard probabilistic modelchecking techniques. If, in fact, $\mathcal{M} \models \varphi$, then there exists no minimal critical edge set and we can directly abort the search. To ease the presentation, we require $\mathcal{M}$ to have exactly one initial state, i. e., $S^0 = \{s^0\}$, and that $\mathfrak{N}$ is *canonically synchronizing*. The latter means that for each action $\alpha \neq \tau$, every automaton $\mathfrak{A}_i$ that possesses the action α ($\alpha \in Act_i$) needs to join the synchronization. This formally amounts to

$$Syn = \left\{ \langle \alpha_1, \ldots, \alpha_n, \alpha \rangle \mid \alpha \in Act \wedge \alpha_i = \begin{cases} \alpha & \text{if } \alpha \in Act_i \\ \bot & \text{otherwise} \end{cases} \right\}.$$

Note that these restrictions can be lifted and do not affect the applicability of the approach in general. Most importantly, we from now on assume a labeling

$$L\colon S \times Act \times Dist(S) \to \mathscr{P}(E(\mathfrak{N}))$$

of the choices in $\mathcal{M}$ with the edges that were involved in generating them according to the semantics of $\mathfrak{N}$ (see Definition 24 and Definition 27). Such a labeling can easily be built during model construction. In practice, it could be that the same choice is generated by multiple edge combinations. However, this is a technicality and can easily be circumvented by relabeling choices uniquely.

Using L, we define the *sources* and *destinations* of an edge set E as follows.

Definition 31 (Edge Sources and Destinations)**.** For a set E of edges, we let

$$src_T(E) = \left\{ s \models_{\mathcal{M}} \exists\lozenge T \mid s \xrightarrow{\alpha} \mu \wedge E = L(s, \alpha, \mu) \right\}$$

$$dst_T(E) = \left\{ s' \models_{\mathcal{M}} \exists\lozenge T \mid \exists s \in S \,.\, s \xrightarrow{\alpha} \mu \wedge s' \in supp(\mu) \wedge E = L(s, \alpha, \mu) \right\}$$

be the sources and destinations of E, respectively.

These are all states that can reach the target set T and are source or destination, respectively, of a distribution that is labeled with E. As we fix the set of target states over the whole chapter, we will omit the subscript for better readability. As a special case, we let $src(e) = src(\{e\})$ and $dst(e) = dst(\{e\})$.

Example 24. Reconsider Example 23 and in particular the PA in Figure 4.3. For $T = \{s_4, s_5, s_6\}$, we have $src(e_2) = \{s_1, s_2\}$, $dst(e_2) = \{s_2, s_3\}$ and $dst(\{e_0, e_1\}) = \{s_1\}$. Note that $dst(e_2)$ does not include the states s_7 and s_8, because these states cannot reach the target set.

4.2 Enumerative Computation of Minimal Critical Edge Sets

The basic idea of our algorithm is very simple. In a nutshell, it enumerates subsets of edges of $E(\mathfrak{N})$ of increasing size, starting with all candidate edge sets E of size 0, 1, 2, and so on. For every enumerated set E, we use a model checker to determine whether $\mathcal{M}|_E$ violates φ. If so, E is a minimal critical edge set by construction. If not, the enumeration process is continued.

Clearly, this enumeration is infeasible for reasonable models: if the automata of $\mathfrak{N}$ altogether possess k edges, in the worst case all possible 2^k edge sets are enumerated, which happens if all edges are in fact necessary to refute φ. Therefore, it is crucial to consider additional information to make the search more efficient. To this end, we prune the search space by introducing a *set of logical constraints* Φ over Boolean variables $\Phi_E = \{x_e \mid \exists i \in \{1, \dots, k\} \,.\, e \in E_i\}$. These constraints, which we treat in detail in Section 4.3, describe properties of all minimal critical edge sets of $\mathfrak{N}$ and therefore do not rule out any optimal solution. Initially, only constraints that can be statically derived from $\mathfrak{N}$ are contained in Φ. We then employ a MaxSat solver to compute the smallest set of edges that is in compliance with Φ as $E = \textsc{MinSat}(\Phi_E, \Phi)$. [1]

Consequently, E is the smallest set of edges that is a viable candidate set for a minimal critical edge set, provided no solution has been found yet. We then construct the restricted model $\mathcal{M}|_E$ and query a standard probabilistic model checker whether $\mathcal{M}|_E \models \varphi$, or, equivalently, $\mathrm{Pr}^{+}_{\mathcal{M}|_E}(\Diamond T) \leq \lambda$. If not, the edge set is sufficient for the construction of a high-level counterexample. If, however, E is not yet sufficient, the enumeration process needs to be continued. One way would be to assert in Φ the formula

$$\bigwedge_{e \in E} x_e \rightarrow \bigwedge_{e \in E(\mathfrak{N}) \setminus E} x_e$$

to enforce that E will not be enumerated again. However, this amounts to an unguided search as only the information that E is insufficient is captured by it. We therefore strive

[1] Formally, this is not entirely correct. First, since MinSat returns a set of assignments, we pick an arbitrary single element. Secondly, the assignment induces an edge set via $E = \{e \mid v(x_e) = true\}$ where $v = \textsc{MinSat}(\Phi_E, \Phi)$.

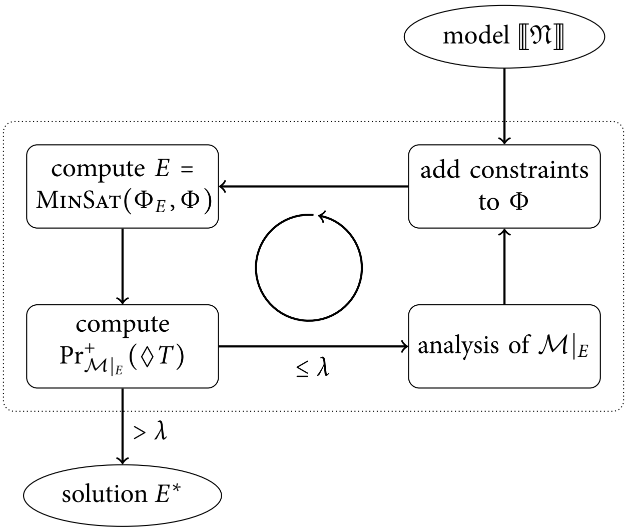

model $[\![\mathfrak{N}]\!]$
add constraints to Φ
compute E = $\textsc{MinSat}(\Phi_E, \Phi)$
compute $\Pr^{+}_{\mathcal{M}|_E}(\lozenge T)$
$\leq \lambda$
analysis of $\mathcal{M}|_E$
$> \lambda$
solution E^*

Figure 4.4: A schematic overview of our MaxSat-based approach.

to analyze $\mathcal{M}|_E$ and derive explanations in the form of additional constraints why E is insufficient, which are then added to Φ. A schematic overview of this procedure is depicted in Figure 4.4. It is not hard to see (Section 4.5) that the algorithm is sound and complete provided that all constraints do not rule out minimal critical edge sets. Formally, the constraint system Φ is sound, if for all *minimal* critical edge sets E^*

$$\nu_{E^*} \models \Phi \quad \text{where} \quad \left(\nu_{E^*}(x_e) = \mathit{true} \iff e \in E^*\right). \tag{4.1}$$

All constraints that we present in the following maintain this property.

4.3 Pruning the Search Space

As previously mentioned, the constraints are the key element in our enumeration procedure. For their derivation, it suffices to consider the fragment of the state space of $\mathcal{M}$ that can reach the target set T, which we refer to as the *relevant fragment*. Throughout this section, we will use E^* to denote a minimal critical edge set and E as a candidate for a solution.

Certain Edges. For typical models, some edges need to be taken along all paths from the initial state to a target state. It is therefore beneficial to determine this set in a preprocessing step of the actual enumeration and thereby possibly prune large parts of the search space. For a given state $s \in S$ and a set of states $S' \subseteq S$, we let

$$\mathcal{C}^{\mathcal{M}}_{s,S'} = \bigcap_{\pi \in \Diamond S'(s,\mathcal{M})} E_{S'}(\pi)$$

be the set of *certain edges* that are taken along all paths from s to S' where

$$E_{S'}(\pi = s_0 \xrightarrow[\mu_0]{\alpha_0} s_1 \xrightarrow[\mu_1]{\alpha_1} s_2 \ldots) = \bigcup_{0 \le i < \min \pi @ S'} L(s_i, \alpha_i, \mu_i) \text{ with } \pi @ S' = \{ i \in \mathbb{N} \mid s_i \in S' \} .$$

This set can be computed using a standard fixed-point analysis [NNH99] on $\mathcal{M}$. Initially, we can assert

$$\bigwedge_{e \in \mathcal{C}^{\mathcal{M}}_{s^0,T}} x_e . \tag{4.2}$$

Example 25. Reconsider the PA $\mathcal{M}$ from Example 23. All paths from the initial state to one of the target states must take the first choice labeled with $\{e_0, e_1\}$. Clearly, any critical edge set must contain them and the constraint system Φ can be strengthened by including

$$x_0 \wedge x_1 .$$

Synchronization Implications. By the semantics of the NSPA $\mathfrak{N}$ and our simplifying assumption that $\mathfrak{N}$ is canonically synchronizing (see Section 4.2), an edge e of the SPA $\mathfrak{A}_i$ that is labeled with action $act(e) = \alpha \neq \tau$ can only generate a choice together with edges from automata that need to synchronize on α. We can conclude that all minimal critical edge sets either do not include e or take at least one edge labeled with α from every $\mathfrak{A}_j$ with $\alpha \in Act_j$. However, this is rather coarse as these additional edges might not be able to synchronize with e because their guards prevent them from being enabled at a common state. As we assume to have $\mathcal{M}$ available, we can be more precise: instead of requiring edges that *potentially* synchronize with e to generate choices, we can require edges that certainly do so in the relevant fragment of $\mathcal{M}$. We assert this as

$$x_e \to \bigvee_{s \in src(e)} \bigvee_{\substack{\langle \alpha,\mu \rangle \in \Delta(s) \\ e \in L(s,\alpha,\mu)}} \bigwedge_{\substack{e' \in L(s,\alpha,\mu) \\ e \neq e'}} x_{e'} \text{ for all } \mathfrak{A}_i, e \in E_i . \tag{4.3}$$

Example 26. In the NSPA $\mathfrak{N}$ from Example 23, the automata need to synchronize along the edges labeled with *msg*. The synchronization implications

$$x_0 \rightarrow x_1 \text{ and } x_1 \rightarrow x_0 \tag{4.4}$$

ensure that enumerated sets must either contain both or none of the two edges.

Forward and Backward Implications. Suppose that an edge $e \in E$ only participates in generating choices that lead to non-target states without outgoing choices in the restricted model $\mathcal{M}|_E$, i. e.

$$e \in L(s,\alpha,\mu) \implies \forall s' \in supp(\mu) \left(s' \notin T \wedge (s' \xrightarrow{\alpha} \mu \implies L(s',\alpha,\mu) \notin E)\right).$$

Then, no path from the initial to a target state in $\mathcal{M}|_E$ uses a choice $\langle s,\alpha,\mu\rangle$ with $e \in L(s,\alpha,\mu)$ and, hence, e can be omitted from E without affecting the reachability probability. Thus, we can conclude that e either generates at least one choice (i) with a target state as a successor or (ii) has some successor state s with a non-empty choice set $\Delta|_E(s)$. We therefore add the constraints

$$x_e \rightarrow \bigvee_{s' \in dst(e)} \bigvee_{\substack{\langle\alpha,\mu\rangle\in\Delta(s')\\ L(s,\alpha,\mu)\neq\{e\}}} \bigwedge_{e'\in L(s',\alpha,\mu)} x_{e'} \text{ for all } \mathfrak{A}_i, e \in E_i, dst(e) \cap T = \varnothing \tag{4.5}$$

to Φ, which we refer to as *forward implications*.

Example 27. Reconsider the PA $\mathcal{M}$ from Example 23. No choice labeled with $\{e_2\}$ directly leads to a target state. Consequently, including e_2 has no effect on the reachability probability unless the label set of some choice in some state in $dst(e) = \{s_2, s_3\}$ is also included. Hence, we add the constraint

$$x_2 \rightarrow x_3 \vee x_4. \tag{4.6}$$

Note that the constraints (4.5) consider every edge *individually*, which might be coarser than necessary. For example, there might be a single choice $\langle s,\alpha,\mu\rangle$ in $\mathcal{M}$ labeled with $\{e_1, e_2\}$ such that $supp(\mu) \cap T \neq \varnothing$, which precludes asserting the constraints (4.5) for e_1 and e_2. However, a solution to the constraint system might only set one of the variables x_{e_1} and x_{e_2} to true and the other one to false and therefore not have the choice $\langle s,\alpha,\mu\rangle$ in the restricted model. We therefore generalize the forward implications to synchronizing

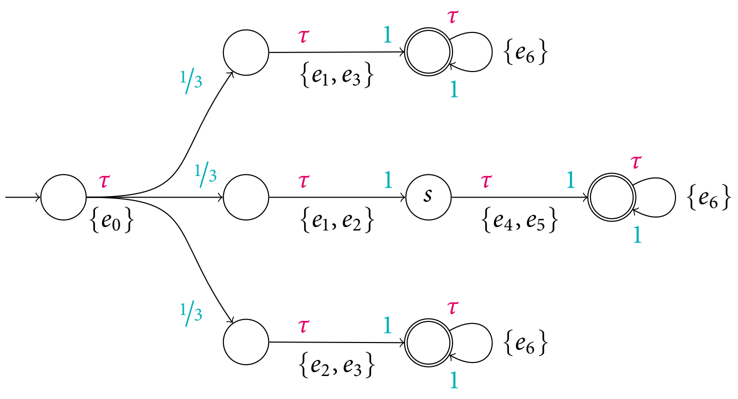


Figure 4.5: A counterexample for the simple generalized forward implications.

sets of edges. An initial attempt is to assert a straight adaptation of constraints (4.5):

$$\bigwedge_{e \in E'} x_e \rightarrow \bigvee_{s' \in dst(E')} \bigvee_{\substack{\langle \alpha,\mu \rangle \in \Delta(s) \\ L(s,\alpha,\mu) \neq E'}} \bigwedge_{e' \in L(s',\alpha,\mu)} x_{e'} \qquad \begin{array}{l}\text{for } E' \subseteq E(\mathfrak{N}) \text{ such that} \\ \exists s, \alpha, \mu \,.\, L(s,\alpha,\mu) = E' \\ \text{and } dst(E') \cap T = \varnothing.\end{array} \tag{4.7}$$

As it turns out, these constraints are not sound and there are minimal critical edge sets that violate them.

Example 28. Consider the PA $\mathcal{M}$ in Figure 4.5 together with the safety property $\mathsf{P}_{\leq \frac{1}{2}}(\Diamond T)$ where the target states are indicated by double circles. Clearly, the property is violated as the maximal reachability probability is 1. The only minimal critical edge set is $E^* = \{e_0, e_1, e_2, e_3\}$, which induces a restricted model $\mathcal{M}|_{E^*}$ with a reachability probability of $\frac{2}{3}$ as both the upper and the lower paths are included. The generalized constraints (4.7) would, however, assert

$$x_1 \wedge x_2 \rightarrow x_4 \wedge x_5 \tag{4.8}$$

because they require the choice labeled with $\{e_1, e_2\}$ to have a successor choice at s. The implication can therefore not be sound as $e_1, e_2 \in E^*$, but $e_4, e_5 \notin E^*$.

Intuitively, the reason why these constraints are unsound is that non-synchronizing edges are always included because of the choices they generate *on their own* and should therefore only be included if they have any possibility to proceed towards a target state. In contrast, a set of synchronizing edges $E' \subseteq E^*$ may be fully included in a minimal critical edge set E^*, but no choice labeled with E' is necessary to exceed the probability bound (see Example 28) or such choices are not even reachable in the restricted model $\mathcal{M}|_{E^*}$. In these cases, the choices are superfluous and we must not require that they have enabled successor edge sets. Instead, we can show the following constraint to be sound that extends the original constraints with a disjunct $\Phi_{Syn}(E')$:

$$\bigwedge_{e \in E'} x_e \rightarrow \left(\bigvee_{s' \in dst(E')} \bigvee_{\substack{\langle \alpha, \mu \rangle \in \Delta(s') \\ L(s,\alpha,\mu) \neq E'}} \bigwedge_{e' \in L(s',\alpha,\mu)} x_{e'} \right) \vee \Phi_{Syn}(E) \quad \begin{array}{l} \text{for } E' \subseteq E(\mathfrak{N}) \text{ such that} \\ \exists s, \alpha, \mu \,.\, L(s,\alpha,\mu) = E' \\ \text{and } dst(E') \cap T = \varnothing. \end{array} \tag{4.9}$$

where

$$\Phi_{Syn}(E') = \begin{cases} \bigwedge_{e \in E'} \Phi_{Syn}(e, E') & \text{if } |E'| > 1 \\ \mathit{false} & \text{otherwise} \end{cases} \tag{4.10}$$

$$\Phi_{Syn}(e, E') = \bigvee_{s \in src(e)} \bigvee_{\substack{\langle \alpha, \mu \rangle \in \Delta(s) \\ e \in L(s,\alpha,\mu) \neq E'}} \bigwedge_{\substack{e' \in L(s,\alpha,\mu) \\ e' \neq e}} x_{e'}. \tag{4.11}$$

With $\Phi_{Syn}(E')$ encoding the fact that the edges $e \in E'$ all appear in enabled synchronizing combinations *different* from E', this weakens the previous assertions (4.7) and allows that the subset E' is included to enable other synchronizations in the system.

Example 29. Coming back to the PA $\mathcal{M}$ of Example 28, the corrected constraints (4.9) simplify to

$$x_1 \wedge x_2 \rightarrow (x_4 \wedge x_5) \vee x_3 \tag{4.12}$$

which does not contradict the optimal solution $E^* = \{e_0, e_1, e_2, e_3\}$.

Similar to forward implications, we can derive *backward implications* with the idea that every selected edge generates at least one choice from some state s such that s is in the support of some choice that is generated by the current set of edges. Formally, for each edge $e \in E$ that is not enabled in the initial state, we select a combination of edges that leads to some state $s \in src(e)$ by enforcing

$$x_e \to \bigvee_{s \in src(e)} \bigvee_{s' \in pred(s)} \bigvee_{\substack{\langle \alpha,\mu \rangle \in \Delta(s') \\ L(s,\alpha,\mu) \neq \{e\} \\ s \in supp(\mu)}} \bigwedge_{e' \in L(s',\alpha,\mu)} x_{e'} \text{ for } \mathfrak{A}_i, e \in E_i, s^0 \notin src(e) \tag{4.13}$$

As before, the constraints can be lifted to reason about synchronizing sets of edges and we can assert

$$\bigwedge_{e \in E'} x_e \to \bigvee_{s \in src(E')} \bigvee_{s' \in pred(s)} \bigvee_{\substack{\langle \alpha,\mu \rangle \in \Delta(s) \\ L(s,\alpha,\mu) \neq E' \\ s \in supp(\mu)}} \bigwedge_{e \in L(s',\alpha,\mu)} x_e \vee \Phi_{Syn}(E') \tag{4.14}$$

for $E' \subseteq E(\mathfrak{N})$ such that $\exists s, \alpha, \mu \; L(s,\alpha,\mu) = E'$ and $s^0 \notin src(E)$. As slight variations of these implications, we can encode that at least one of the choices of the initial state and at least one choice that has a target state as a direct successor are generated by minimal critical edge sets.

Example 30. Reconsider our running Example 23. In a backward manner, we assert that including edge e_2 is only reasonable if either $\{e_0, e_1\}$ or e_3 is also taken, because only via these edges a state can be reached that has a choice generated by e_2. In the form of the constraints (4.13), this is formulated as

$$x_2 \to (x_0 \wedge x_1) \vee x_3. \tag{4.15}$$

Extended Backward Implications. As argued in Example 30, including e_2 in a solution must trigger taking $\{e_0, e_1\}$ or e_3. However, we can be more precise. As e_2 represents an attempt at processing the received message, it is evident that this can only happen after a message has been received and we would therefore like to assert

$$x_2 \to x_0 \wedge x_1, \tag{4.16}$$

which is stronger than the constraint obtained in (4.13). Intuitively, the set $\{e_0, e_1\}$ enables choices labeled with $\{e_2\}$, because it takes the system from state s_0 in which there is no choice labeled with $\{e_2\}$ to state s_1 in which there is such a choice. Note that this is not the case for e_3. More formally, we say that a set of edges E'' *enables* an edge set E' that is not enabled in the initial state, i. e., $s^0 \notin src(E')$, if there is at least one state s such that

(i) $s \notin src(E')$, and

(ii) $s \xrightarrow{\alpha} \mu$ with $L(s,\alpha,\mu) = E''$ and for some successor state $s' \in supp(\mu)$, $s' \in src(E')$.

In this case, E'' is necessary to establish the conditions for E' to be enabled. Let $enable(E')$ denote all sets of edges that enable E'. We can then assert

$$\bigwedge_{e\in E'} x_e \rightarrow \bigvee_{E''\in enable(E')} \bigwedge_{e\in E''} x_e \vee \Phi_{Syn}(E') \qquad \begin{array}{l}\text{for } E' \subseteq E(\mathfrak{N}) \text{ such that} \\ \exists s,\alpha,\mu \,.\, L(s,\alpha,\mu) = E' \\ \text{and } dst(E') \cap T = \emptyset.\end{array} \tag{4.17}$$

without ruling out optimal solutions.

Enforcing Reachability of a Target State. Despite all previously presented constraints, the solver potentially enumerates edge sets that have no path from the initial to a target state. Such edge sets can never serve as a minimal critical edge set and we would therefore like to avoid enumerating them. Using a similar construction as the one in [Wim+15], reachability of a target state can be encoded using additional constraints. This comes at the cost of a significant increase in the number of constraints. It also requires the use of a MaxSmt solver (as opposed to a MaxSat solver) as the constraints make use of real-valued variables $\Phi_r = \{r_s \mid s \in S\}$ in addition to further Boolean variables $\Phi_t = \{t_{s,s'} \mid s,s' \in S\}$. The constraints

$$\bigvee_{s'\in succ(s^0)} t_{s^0,s'} \tag{4.18}$$

$$\bigvee_{s'\in pred(s)} t_{s',s} \implies \bigvee_{s''\in succ(s)} t_{s,s''} \text{ for } s \in S \setminus T \tag{4.19}$$

$$t_{s,s'} \implies \bigvee_{\substack{\langle\alpha,\mu\rangle\in\Delta(s)\\ s'\in supp(\mu)}} \bigwedge_{e\in L(s,\alpha,\mu)} x_e \text{ for } s,s' \in S \tag{4.20}$$

$$t_{s,s'} \implies r_s < r_{s'} \text{ for } s,s' \in S \tag{4.21}$$

enforce that a target state is reachable from the initial state. Intuitively, they require the solver to synthesize a chain of transitions (constraints (4.19)) from the initial state (constraints (4.18)) along which

(i) the labeling of the choices associated with these transitions is included (constraints (4.20)), and

(ii) (strict) progress towards the target states is made through the strict ordering imposed on the variables from Φ_r (constraints (4.21)).

Together, this amounts to the existence of a loop-free path from an initial to a target state that is fully contained in the restricted model. Note that the (expensive) constraints (4.21) are in fact necessary as otherwise a lasso-shaped path (a stem that passes into a loop) that never reaches T would be an admissible solution.

4.4 Guiding the Enumeration

After the initial constraint set was constructed, a MinSat problem is solved to obtain a smallest edge set E that adheres to these constraints. The restricted model $\mathcal{M}|_E$ is then dispatched to a model checker to verify or refute φ. If the reachability probability in $\mathcal{M}|_E$ exceeds λ, a solution for the minimal critical edge set problem has been found, because the set E is, by construction, the smallest candidate set. However, in the more likely event of not exceeding λ, we aim to derive additional constraints from the constrained model that guide the solver towards a solution with a higher reachability probability. While it is possible to rule out just the current (insufficient) candidate set E by adding

$$\bigwedge_{e \in E} x_e \to \bigwedge_{e \in E(\mathfrak{N}) \setminus E} x_e$$

to Φ, we strive to rule out more insufficient edge sets to guide the search. For the sake of illustration, suppose that for the most recently enumerated set of edges E cannot reach T, i. e. $\Pr^{+}_{\mathcal{M}|_E}(\Diamond T) = 0$. Then, there is not a single path from the initial to the target states in the restricted model $\mathcal{M}|_E$. That means that the states $\overleftarrow{E}$ that are reachable from the initial state in $\mathcal{M}|_E$ are disconnected from the states $\overrightarrow{E}$ that can reach a target states in $\mathcal{M}|_E$. In order to achieve a non-zero reachability probability, any extension of E that is enumerated, must try to "connect" $\overleftarrow{E}$ and $\overrightarrow{E}$ in the sense that they must establish a path from some state $s \in \overleftarrow{E}$ to some state $s' \in \overrightarrow{E}$. We consider the states on the *border of* E, defined as

$$E\!\uparrow\; = \Big\{ s \in \overleftarrow{E} \mid s \xrightarrow{\alpha} \mu \text{ with } L(s, \alpha, \mu) \notin E \\ \wedge\; \exists s' \in \mathit{supp}(\mu) \text{ with } s' \notin \overleftarrow{E} \wedge s' \models \exists \Diamond \overrightarrow{E}(\mathcal{M}) \Big\}$$

that have (i) some choice that is not contained in the restricted model (ii) that has at least one successor state that is not part of the restricted model and (iii) can reach $\overrightarrow{E}$ (in the original PA $\mathcal{M}$). This set can be efficiently obtained using graph searches.

Clearly, a path from $\overleftarrow{E}$ to $\overrightarrow{E}$ must visit some state $s \in E\!\uparrow$. In particular, such a path must include all certain edges $\mathcal{C}^{\mathcal{M}}_{s,\overrightarrow{E}}$. We can therefore assert

$$\bigwedge_{e \in E} x_e \rightarrow \bigvee_{s \in E\uparrow} \bigwedge_{e \in \mathcal{C}^{\mathcal{M}}_{s,\overrightarrow{E}}} x_e \tag{4.22}$$

However, it may be the case that the sets of certain edges $\mathcal{C}^{\mathcal{M}}_{s,\overrightarrow{E}}$ are empty for all states $s \in E\!\uparrow$. In this case, the constraint (4.22) does not add any information and does not provide guidance to the solver. Even worse, it does not even rule out E and as the solver is free to enumerate the same solution again, there is no progress towards the solution. However, we observe that from one of the border states, we must "cross" the latter and take a choice $\langle s, \alpha, \mu \rangle$ to a state that is currently not reachable in the restricted model $\mathcal{M}|_E$. That means $L(s, \alpha, \mu) \nsubseteq E$ and we need to included the missing labels to enable this choice. We therefore extend the constraint (4.22) by an intermediate step to a previously unreachable state and obtain the new constraint

$$\bigwedge_{e \in E} x_e \rightarrow \bigvee_{s \in E\uparrow} \bigvee_{\substack{\langle \alpha,\mu \rangle \in \Delta(s) \\ supp(\mu) \nsubseteq \overleftarrow{E}}} \bigvee_{\substack{s' \in supp(\mu) \setminus \overleftarrow{E} \\ s' \models_{\mathcal{M}} \exists \Diamond T}} \left(\underbrace{\bigwedge_{e \in L(s,\alpha,\mu)} x_e}_{s \rightarrow s' \notin \overleftarrow{E}} \wedge \underbrace{\bigwedge_{e' \in \mathcal{C}^{\mathcal{M}}_{s',\overrightarrow{E}}} x_{e'}}_{s' \rightarrow \overrightarrow{E}} \right) \tag{4.23}$$

Example 31. Reconsider the running Example 23. Assume that the solver found the candidate edge set $E = \{e_0, e_1, e_2\}$. Note that this is not possible if all previously presented constraints are added, but is assumed here for the sake of illustration. The set of states that is reachable in the restricted model is $\overleftarrow{E} = \{s_0, s_7, s_8, s_1, s_2, s_3\}$, the latter three of which are on the border $E\!\uparrow$. The constraints (4.23) now simplify to

$$x_0 \wedge x_1 \wedge x_2 \rightarrow x_3 \vee x_4.$$

The constraints (4.23) only apply to the case where the maximal reachability probability in the restricted model is zero. They ensure that at least one state that previously was not reachable from the initial state will be part of $\overleftarrow{E'}$ for any edge set $E' \supseteq E$. This is, however, no longer the case if which the maximal reachability probability is greater than zero in the restricted model $\mathcal{M}|_E$. This is because new edges may increase the reachability probability by (directly) connecting states that are already reachable with a higher probability. Hence, in this case we weaken the constraints (4.23) to

$$\bigwedge_{e \in E} x_e \rightarrow \bigvee_{s \in \overleftarrow{E}} \bigvee_{\langle \alpha, \mu \rangle \in \Delta(s)} \bigvee_{\substack{s' \in supp(\mu) \\ s' \models_{\mathcal{M}} \exists \Diamond T}} \left(\underbrace{\bigwedge_{e \in L(s, \alpha, \mu)} x_e}_{s \rightarrow s'} \wedge \underbrace{\bigwedge_{e' \in \mathcal{C}^{\mathcal{M}}_{s, \overrightarrow{E}}} x_{e'}}_{s' \rightarrow \overrightarrow{E}} \right). \tag{4.24}$$

4.5 Soundness and Completeness

As argued in Section 4.3 and Section 4.4, constraints in Φ do not rule out solutions to the minimal critical edge set problem and property (4.1) is satisfied at all times.

Soundness. Suppose that the algorithm returns a critical edge set E as a result. Now assume that there is a minimal critical edge set E^* with $|E^*| < |E|$. Because of (4.1), we additionally have that $\nu_{E^*} \models \Phi_{\mathfrak{N}}$. As the algorithm enumerates edge sets ordered by their sizes, E^* is enumerated before E and the algorithm therefore had returned E^* as the result, which contradicts that the algorithm returned E.

Completeness. Assume that there is a minimal critical edge set E_1^* of size $|E_1^*| = k$. By (4.1), $\nu_{E_1^*} \models \Phi$. Our algorithm enumerates the edge sets as long as there is a solution to Φ. As there are only finitely many edge sets of size at most k, eventually E_1^* (or another critical edge set of size k) is enumerated and returned.

4.6 Evaluation

Implementation. We implemented our algorithm within the framework of Storm (see Chapter 7). We employ a counter-based MaxSat procedure [FM06] using Z3 4.6 [MB08] as the underlying solver.

To provide a fair comparison, we additionally implemented the Milp-based approach [Wim+15] using the commercial solver Gurobi 7.5.2 [Gur16]. We also augment the MILP-based approach with the detection of certain edges (see Section 4.3) as an optimization and add the resulting information to the problem encoding. As proposed in [Wim+15], we added the so-called *scheduler cuts*, an additional set of constraints to rule out suboptimal solutions, to the Milp encoding, because they strongly tend to improve the performance of the solver. According to [Wim+15], all other cuts have a mixed influence on the performance of the solver and were thus omitted.

Experimental results. For the evaluation of the prototype we used the four benchmarks that were considered in [Wim+15]. They are part of Prism's benchmark suite [KNP12]. Both the models and the properties we used are described in more detail in Appendix C. As our approach uses **JANI** models, we used Storm's capability of transforming **PRISM** models to **JANI** (see Section 3.4 in Chapter 3).

Table 4.1 shows different instantiations of the models along with their sizes in terms of states and transitions. For each instance, we give the maximal reachability probability with respect to the target set T and the probability threshold λ that we considered in the probabilistic safety property. In all cases, we chose this bound to be around 50% of the total (maximal) reachability probability. Finally, we show how many edges were necessary to form a high-level counterexample in column $|E^*|$ next to the total number $|E|$ of edges in he model. For the cases where we were unable to compute a high-level counterexample, we indicate the best lower bound we could prove. The ratio of $|E^*|$ to $|E|$ therefore is an indicator of how concisely edge-based high-level counterexamples could capture the essence that explains the violation of the property.

As the table shows, this ratio varies substantially between models. For the `csma` instances, almost all edges are necessary to violate the property. This is despite the fact that the target reachability probability was only 0.5 and the actual reachability probability is almost 1 for the two smaller instances. Slightly better, for the `coin` and `wlan` case studies about 50% to 60% of the edges are sufficient to achieve enough probability mass. Finally, high-level counterexample perform best for the `firewire` models in which almost two thirds of all edges can be deleted without losing violation of the safety property.

As the largest reductions are obtained for the smallest instances of `firewire` and `wlan`, we study the impact of the probability bound λ on the size of minimal critical edge sets. For this, we vary λ and give the sizes of critical edge sets in Figure 4.6 from 0% to 100% of the actual (maximal) reachability probability in the models in steps of 10%. We see that for `wlan(2,2)`, the increase in size of the minimal critical edge sets remains moderate and 100% of the probability can be achieved using only 34 of the 70 edges of the model. In contrast, for the `firewire(3)` instance, going from 20% to 60% requires 8 more edges or an increase in size of 40% and obtaining at least 80% of the overall probability requires at least 38 (of 64) edges.

We now move to a performance comparison of the Milp- and the MaxSat-based approach. For this, we computed high-level counterexamples for all instances using both approaches. The experiments were conducted on a HP BL865C G7 blade with 48 cores clocked at 2.0 GHz and 192GB of RAM running Debian 9.0 (Stretch). We set a timeout of one hour for each individual (single-threaded) experiment. For the MILP approach, we performed experiments with and without using the scheduler cuts and

				probabilities		edges	
model	instance	states	transitions	λ	$\Pr^+(\lozenge T)$	$\lvert E^*\rvert$	$\lvert E\rvert$
`coin`	(2, 2)	272	492	0.30	0.56	8	14
	(4, 4)	43,136	144,352	0.30	0.54	17	28
	(6, 2)	1,258,240	6,236,736	0.30	0.59	≥ 16	42
`csma`	(2, 4)	7,958	10,594	0.50	> 0.99	36	38
	(2, 6)	66,718	93,072	0.50	> 0.99	36	42
	(4, 2)	761,962	1,327,068	0.40	0.78	≥ 43	72
`firewire`	(3)	4,093	5,585	0.50	1	24	64
	(12)	22,852	40,904	0.50	1	24	64
	(36)	212,268	481,792	0.50	1	24	64
`wlan`	(2, 2)	28,598	57,332	0.10	0.18	33	70
	(4, 4)	345,120	762,422	4e−4	7.9e−4	39	76
	(6, 6)	5,007,670	11,475,920	1e−7	2.2e−7	43	80

Table 4.1: The model and counterexample sizes for the benchmark models.

report on the best of these results. Encoding reachability of a target state as presented in Section 4.3 tended to be too expensive for the MaxSat solver. It slowed down enumerating the candidate edge sets so much that this could not be made up for by the reduced number of model checking calls that needed to be performed. Thus, we list the times obtained without adding these constraints.

Table 4.2 summarizes the results of our experiments. For each considered model instance, we give the runtime in seconds and memory consumption in gigabytes of both approaches. If an experiment timed out, we indicate this with TO.

First of all, we observe that the Milp approach returns a wrong result for the largest `wlan` instance. It returns an edge set of size 30 that is not sufficient to prove violation of the property. After careful inspection of the encoding, we believe this to be due to imprecisions within Gurobi, which uses numerical procedures rather than exact arithmetic.[2] In particular, it allows that integer variables to have a non-integral value if the deviation is bounded by a given tolerance. Setting this to the lowest possible value (10^{-9}) did not solve the problem. We tried to verify our suspicion and used Z3's (MI)LP

[2]see `http://www.gurobi.com/documentation/7.5/refman/variables_and_constraints.html`

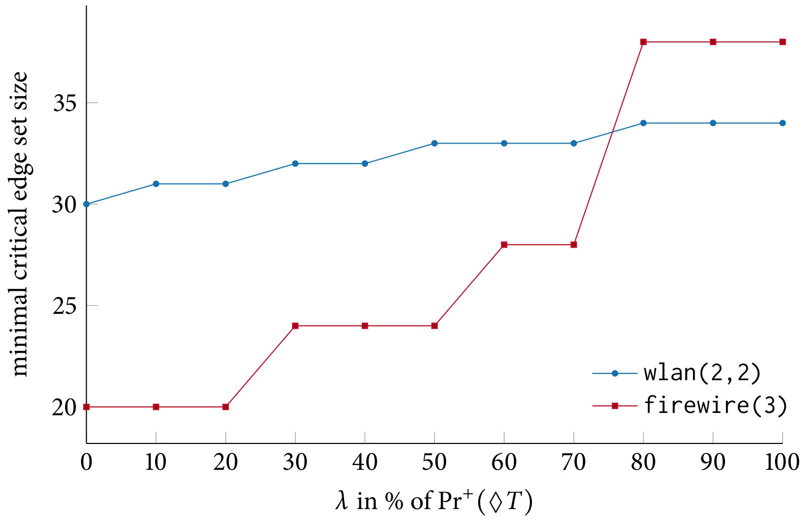


Figure 4.6: Influence of λ for `firewire(3)` and `wlan(2,2)`.

solver that uses rational arithmetic. However, we had to abort the experiment when it consumed more than 140GB of RAM and did not finish within a week.

Next, we notice that the MaxSat approach outperforms the Milp encoding substantially in both time and memory. While Milp times out on the largest instances of all selected case studies, MaxSat solves two of them within the time limit. In particular, the MaxSat approach determines that for the largest `wlan` instance with more than five million states, 43 out of the 80 edges are necessary to prove violation. Checking the induced submodel revealed that — in contrast to the incorrect Milp solution — these edges indeed form a counterexample.

We now turn to the more detailed performance characteristics of our MaxSat approach. The core of the algorithm can be divided into three parts: (i) derivation of constraints (both a priori and in each iteration), (ii) solving the MaxSat problem to determine the next candidate edge set and (iii) the model checking calls to determine whether the current candidate is sufficient. Table 4.3 breaks down the total time consumed by the MaxSat approach according to these three ingredients, where we additionally distinguish the time needed for derivation of the initial constraints (see Section 4.3) and the computation of the dynamic constraints (see Section 4.4). The percentages in the

		Milp		MaxSat	
model	instance	time	memory	time	memory
`coin`	(2, 2)	102.84	0.07	0.05	0.05
	(4, 4)	TO	> 1.00	767.81	0.09
	(6, 2)	TO	> 11.54	TO	> 1.11
`csma`	(2, 4)	89.28	0.11	2.58	0.05
	(2, 6)	189.29	0.48	81.29	0.09
	(4, 2)	TO	> 7.77	TO	> 0.68
`firewire`	(3)	49.18	0.14	7.43	0.06
	(12)	TO	> 0.92	27.60	0.07
	(36)	TO	> 6.91	281.24	0.25
`wlan`	(2, 2)	TO	> 2.13	1.67	0.05
	(4, 4)	TO	> 1.82	40.00	0.07
	(6, 6)	error	error	458.35	0.32

Table 4.2: Performance comparison of the Milp- and MaxSat approaches.

table are to be understood as the fraction of the runtime that was spent on a particular task where the runtime does not include the model building times.

We observe that both the derivation of (dynamic) constraints and the model checking calls dominate the runtime for at least one instance whereas solving the MaxSat problem and the derivation of the initial constraints have a low impact on the total running times. For the `coin(4,4)` instance, the numerical solution of the enumerated submodels takes three times as long as the derivation of all constraints combined. In contrast, model checking the `firewire` case study submodels takes only one tenth of the time needed to derive the constraints. Finally, in the case of `wlan`, both parts contribute roughly equally to the overall runtime. We want to highlight that the overwhelming time dedicated to the derivation of constraints is actually spent on the constraints that dynamically guide the solver in each iteration rather than the ones derived initially. Depending on the model, the fixpoint iteration to derive the certain edges (see Section 4.3) takes a substantial number of iterations.

While this seems very costly, we will now show that the dynamic constraints play a crucial role in eliminating a vast number of candidate edge sets. In Table 4.4 we show

		constraints			
model	instance	initial	dynamic	solving	checking
coin	(2, 2)	53.1%	18.4%	14.3%	14.3%
	(4, 4)	0.2%	22.8%	1.1%	75.9%
csma	(2, 4)	4.3%	58.4%	5.0%	32.3%
	(2, 6)	1.3%	75.7%	0.8%	22.2%
firewire	(3)	1.6%	33.7%	56.0%	8.6%
	(12)	1.2%	69.6%	15.4%	13.8%
	(36)	1.1%	83.1%	1.5%	14.3%
wlan	(2, 2)	5.0%	46.4%	18.0%	30.7%
	(4, 4)	0.5%	54.9%	4.0%	40.6%
	(6, 6)	0.3%	53.5%	0.3%	46.0%

Table 4.3: Time breakdown of the MaxSat approach for the benchmark models.

the runtimes of the MaxSat-based approach with and without the dynamic constraints. With the exception of the `coin` instances, not including them significantly increases the runtimes and leads to a number of time outs. Even though the time spent on deriving the constraints is reduced, not guiding the search is clearly disadvantageous.

To illustrate the overall effectiveness of all constraints combined, we give for each benchmark instance the number of edge sets that were enumerated in Table 4.5. Since each such set amounts to a model checking call, these two measures coincide. To give an indication of how many sets were enumerated in comparison to the number of sets that would have been enumerated if no constraints were added, we compute the fraction

$$\text{fraction of total} = \frac{\text{sets enumerated}}{2^{|E^*|-1}} \cdot 100.$$

Note that we subtract 1 from the size of the minimal critical edge sets to only count sets that would definitely have to be enumerated. This is because when enumerating the sets of size $|E^*|$ the order of enumeration governs whether a minimal critical edge set is found right away or after $\binom{|E|}{|E^*|}$ sets were enumerated. We account for this by not counting the sets of the target size and observe that the actual savings our constraints achieve are *at least* as high as indicated.

As the data shows, the number of enumerated edge sets remains reasonable across

model	instance	with	without
`coin`	(2, 2)	0.05	0.05
	(4, 4)	767.81	797.21
`csma`	(2, 4)	2.58	TO
	(2, 6)	81.29	TO
`firewire`	(3)	7.43	TO
	(12)	27.60	TO
	(36)	281.24	TO
`wlan`	(2, 2)	1.67	749.04
	(4, 4)	40.00	TO
	(6, 6)	458.35	TO

Table 4.4: Runtimes of MaxSat approach with and without dynamic constraints.

		MaxSat	
model	instance	sets enumerated	fraction of total
`coin`	(2, 2)	25	2.0×10^{1}
	(4, 4)	2025	3.1
`csma`	(2, 4)	148	4.3×10^{-7}
	(2, 6)	380	1.1×10^{-6}
`firewire`	(3)	306	3.6×10^{-3}
	(12)	306	3.6×10^{-3}
	(36)	306	3.6×10^{-3}
`wlan`	(2, 2)	296	6.9×10^{-6}
	(4, 4)	715	2.6×10^{-7}
	(6, 6)	694	1.6×10^{-8}

Table 4.5: Information on how many edge sets were enumerated for each instance.

all instances. For the smallest `coin` instance, only 8 edges are necessary to form a counterexample and therefore the 25 enumerated sets are almost 20% of all sets of size at most 7. However, for the larger models, the constraints rule out a lot of candidate sets. Where for the `firewire` case study roughly 3 millionth of the space is enumerated, for the `wlan` case study only 694 of the possible $2^{42} = 4{,}398{,}046{,}511{,}104$ edge sets are enumerated. That amounts to enumerating 1 out of every 10 billion edge sets.

Finally, we want to shed some light on the performance of the Milp approach. Most Milp solvers use a branch-and-bound technique to solve a problem. Initially, fast heuristics search for a good initial solution that provides an upper bound on the optimal objective value (in the case of a minimizing objective as in our setting). Such a solution amounts to an edge set that is sufficient, but not necessarily of minimal size. Simultaneously, the solver maintains a lower bound on the number of edges that is required to exceed the probability threshold. This lower bound stems from solutions to the relaxed (non-integral) problem and is improved over time. Consequently, unless the solver's heuristics were unable to find any solution at all, it possesses an interval in which the optimal objective value lies. The size of this window is referred to as the *gap*. It is quite common that the solver arrives at a good solution quickly, but takes a very long time to prove optimality. In contrast, the MaxSat procedure only has a lower bound on the number of required edges. We therefore analyzed how the lower and upper (if available) bounds develop over time for several considered instances.

Figures 4.7 to 4.10 show a comparison of the evolution of the obtained lower and upper bounds of the techniques over time for four selected model instances. Here, time 0 refers to the point directly after the model building step that both approaches carry out. The horizontal dashed lines indicate the sizes of minimal critical edge sets. As the MaxSat approach does not have a non-trivial upper bound available, the enumeration process cannot be stopped when the target edge set size has been established as a lower bound, because it still might be a strict lower bound. Therefore, we indicate the termination times of the approaches for the instances with vertical dotted lines.

For the `coin(4,4)` and `wlan(4,4)` instances, the Milp solver does not find any (even non-optimal) solution to the problem and consequently is not able to compute an upper bound on the required number of edges. Also, almost no progress is made on the lower bounds, whereas the MaxSat approach progresses steadily towards the target size. For the `csma(2,6)` example, Gurobi obtains lower and upper bounds after a short (about 10 seconds) phase of precomputation. Then, for a long period of time, no progress is made until after about 180 seconds the lower and upper bounds quickly converge to the target edge set size of 36. The progress of the MaxSat procedure again is more steady due to the guiding constraints. Finally, Gurobi can achieve an upper bound of 24 for

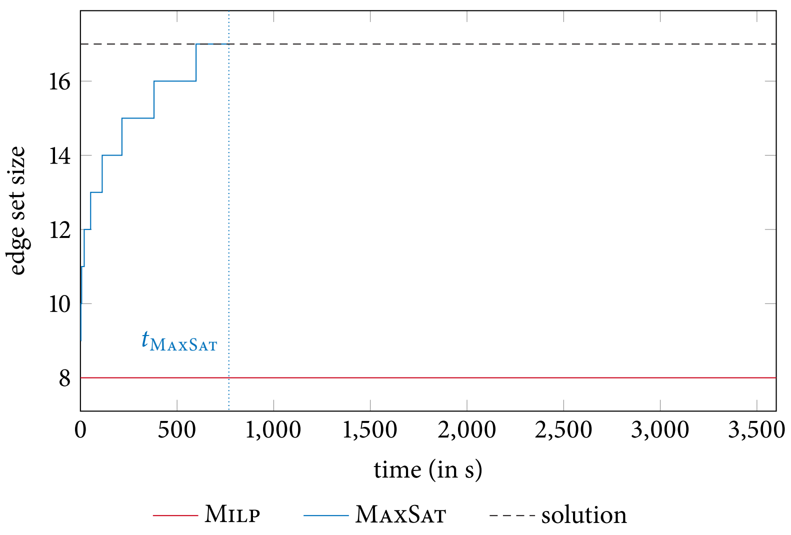


Figure 4.7: Development of bounds over time for `coin(4,4)`.

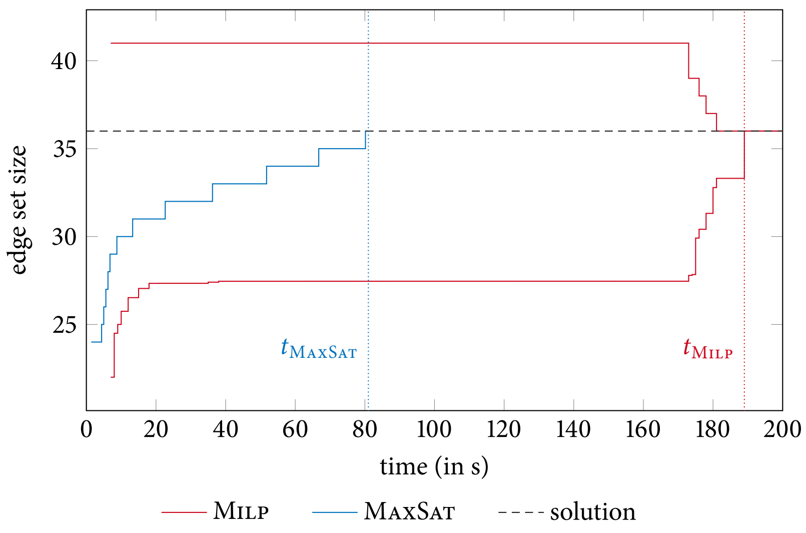


Figure 4.8: Development of bounds over time for `csma(2,6)`.

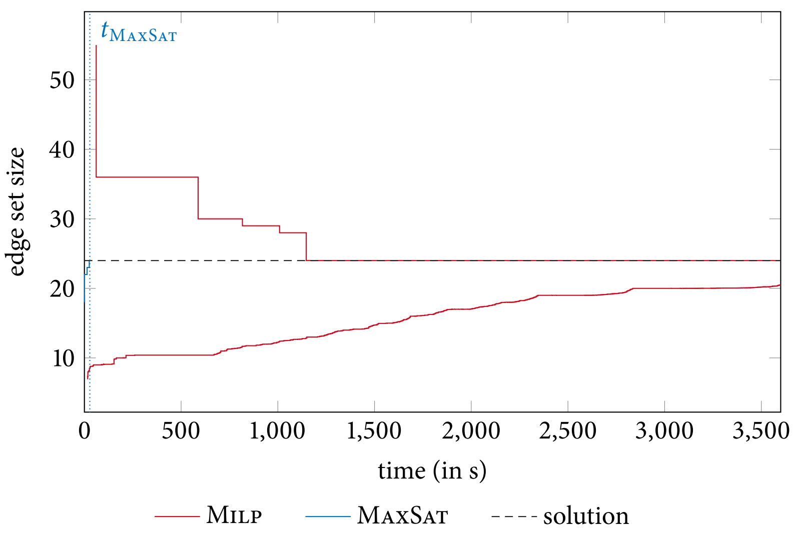


Figure 4.9: Development of bounds over time for `fw(12)`.

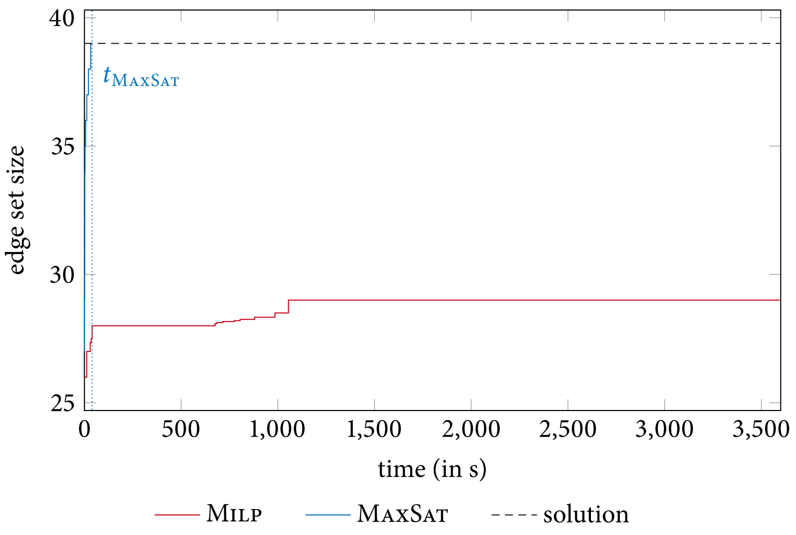


Figure 4.10: Development of bounds over time for `wlan(4,4)`.

the `firewire(12)` benchmark, which happens to be the minimal critical edge set size). It can do so after about 1150 seconds, but afterwards it fails to prove optimality of the solution within the time limit, because the lower bound can only be improved very gradually. In contrast, our MaxSat approach improves the lower bound rapidly and arrives at the solution in under 30 seconds.

Chapter 5

Symbolic Bisimulation Minimization of Markov Automata

5.1 Motivation and Goals

One of the major challenges for automated verification techniques like model checking is the state space explosion problem. The state space of a system grows exponentially in both the number of components and variables. Consequently, real-world systems give rise to a vast state space and cannot be effectively checked without sophisticated abstractions be it either manual or automatic. In the context of probabilistic model checking, this problem becomes even more pressing. The numerical methods for solving linear programs or Bellman equations that are key to the verification of probabilistic systems are inherently computationally more expensive than their qualitative counterparts.

In both the qualitative and quantitative setting, this problem motivated a huge body of research related to coping with large (or even infinite) state spaces in model checking and related techniques. The major commonality between all these approaches is that they represent the state space *symbolically*. For qualitative systems, bounded model checking (BMC) [Bie+03] or IC3 [Bra11] (sometimes also referred to as property-driven reachability or PDR for short) are successful techniques that represent the state space using Boolean variables and logical formulae. Another route is the use of binary decision diagrams (BDDs) for the representation of systems, an approach that in the qualitative case is mostly applied in the verification of hardware circuits. In the probabilistic setting, employing variants of decision diagrams supporting multiple values such as multi-terminal binary decision diagrams (MTBDDs) or multi-valued decision diagrams (MDDs) is the most widely used approach to deal with large state spaces. This is witnessed by implementations in model checkers like Prism, Epmc and Storm (all MTBDDs) and Smart (MDDs). While using MTBDDs enables the storage of enormous models, the numerical solution process typically is slow. To combat this, hybrid techniques [Par03] have been proposed that store parts of the model and solution vectors using MTBDDs and parts using explicit data structures. However, [Par03] argues that to leverage such methods

> »[...] an important focus for future work should be the development of techniques and methodologies for abstraction, which concentrate on reducing the size of model which must be analysed, rather than on finding compact storage for large models. «

The goal of symbolic representations of data structures is to exploit structure in the model and they tend to be small when the model is symmetric. The idea to exploit that many states behave similarly in a real-world model is also the key idea behind *bisimulation minimization*. Instead of trying to analyze the system directly, a smaller variant of it is built that merges all states whose behavior is not distinguishable in a well-defined sense. This potentially smaller *quotient* model preserves the behavior of the original model in the sense that it satisfies the same properties in a fragment of a suitable logic. Therefore, it is sound to analyze the quotient instead of the original model. Bisimulation minimization (BM) was shown to speed up the analysis of Markov chains (MCs) [Kat+07] in many cases when systems are represented *explicitly* in terms of sparse matrices.

Based on the Kannelakis-Smolka algorithm [KS90] to compute bisimulation relations, Blom and Orzan presented a signature-based approach [BO05] that is particularly suited in distributed environments. [Wim10] proposes an elegant algorithm that uses signatures to compute the bisimulation quotient of transition systems and MCs directly on the BDD representation. Using the *combination* of both symbolic representations (DDs) *and* bisimulation minimization, models with billions of states can be reduced to quotient models that are orders of magnitude smaller in size. To obtain these results, the symbiosis of symbolic representation and (symbolic) bisimulation minimization was essential. Without the former, the model could not have been built in the first place and without the latter no reduction would have been possible.

In this chapter, we extend symbolic bisimulation minimization in several directions. First, we show how it can be extended to Markov automata and (by subsumption) probabilistic automata where the main difficulty lies in the mixture of nondeterminism and probabilities. Second, we explain how the procedure can be made aware of rewards in the models. And finally, we accelerate the quotient extraction by deriving its explicit representation directly from the bisimulation partition without the detour over a symbolic representation of the quotient's transition relation, which enables the benefits from both symbolic model minimization and explicit state model checking.

5.2 Bisimulation Equivalence

To reduce the size of the model prior to verification, the most obvious attempt is to merge states. Given a notion of what it means that two states behave equivalently, one can build a *quotient* model that merges equivalent states. Of course, reasonable equivalence relations guarantee that the interesting properties carry over from the quotient to the original model. This enables to minimize the input model with respect to the equivalence and use the quotient model instead of the original model for answering verification queries. One of the most well-studied behavioral equivalences for Markov automata are bisimulation relations. Intuitively, they require related states to be able to mimick one anothers' behavior. Various notions of bisimulation differ on what this means precisely. For example, *strong bisimulation* requires that τ steps need to be mimicked, whereas *weak bisimulation* abstracts from these internal steps. The weaker notions allow for merging more states in the quotient model, but can only preserve the truth values of properties that are oblivious to internal steps and the repetition of atomic proposition sets along a path. In contrast, the stronger notions preserve more properties, but relate fewer states and therefore generate larger quotient state spaces. In addition, strong bisimulation is typically easier and more efficient to compute. For the remainder of this chapter, we focus on strong bisimulation and also refer to it by just bisimulation.

Example 32. As a running example for this chapter, we consider the MA $\mathcal{M}$ in Figure 5.1. Of its four states $S = \{s_0, s_1, s_2, s_3\}$ three are probabilistic (s_0, s_2, s_3) and one one is Markovian (s_1) with exit rate 4. Although not useful in practice, all states are labeled with the empty set for illustration purposes. Unless explicitly stated otherwise, all examples in this chapter refer to this MA.

We start our presentation of bisimulation minimization by defining the quotients of distributions and reward functions under a given equivalence relation.

Definition 32 ((Rewarded) Distribution Quotient)**.** Let $\mu \in Dist(S)$, $\rho\colon S \to \mathbb{R}_{\geq 0}$ a reward function and $\mathcal{R} \subseteq S \times S$ an equivalence relation.

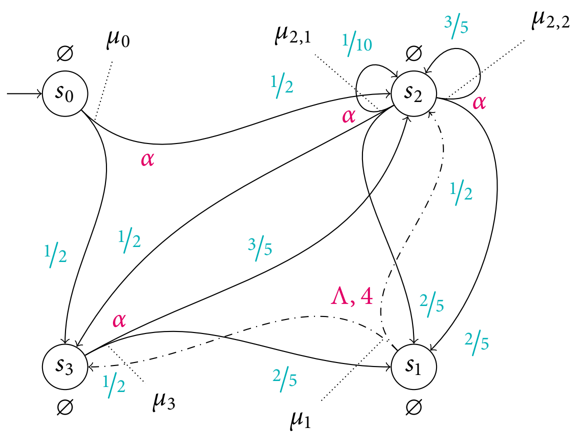


Figure 5.1: The running example MRA $\mathcal{M}$.

- The quotient distribution $[\mu]_\mathcal{R} \in Dist(S/\mathcal{R})$ of μ modulo $\mathcal{R}$ is defined as

$$[\mu]_\mathcal{R}(\mathcal{C}) = \mu(\mathcal{C}) = \sum_{s \in \mathcal{C}} \mu(s)$$

- The quotient reward function $[\rho, \mu]_\mathcal{R}$ of ρ with respect to μ modulo $\mathcal{R}$ is defined by

$$[\rho, \mu]_\mathcal{R} : S/\mathcal{R} \to \mathbb{R}_{\geq 0}$$
$$[\rho, \mu]_\mathcal{R}(\mathcal{C}) = \begin{cases} \sum_{s \in \mathcal{C}} \frac{\mu(s)}{\mu(\mathcal{C})} \cdot \rho(s) & \text{if } \mu(\mathcal{C}) > 0 \\ 0 & \text{otherwise} \end{cases}$$

The quotient of a distribution with respect to an equivalence relation $\mathcal{R}$ on a state space S is again a distribution, but this time on the equivalence classes of $\mathcal{R}$. It assigns to each class $\mathcal{C}$ the sum of probabilities that targets any state in $\mathcal{C}$. Similarly, the quotient reward function assigns rewards to the equivalence classes of $\mathcal{R}$. Before summing the rewards of the states of a class $\mathcal{C}$, they are weighted with the fraction of the probability they are associated with according to μ to preserve their expected values.

Example 33. Let $\mathcal{R}$ be the equivalence relation on $S \times S$ corresponding to the symmetric, transitive and reflexive closure of $\{\langle s_0, s_2\rangle, \langle s_0, s_3\rangle, \langle s_1, s_1\rangle\}$. In other words, the equivalence classes of $\mathcal{R}$ are

$$S/\mathcal{R} = \left\{ \underbrace{\{s_0, s_2, s_3\}}_{C_0^{\mathcal{R}}}, \underbrace{\{s_1\}}_{C_1^{\mathcal{R}}} \right\}.$$

Consider the choice $\langle \alpha, \mu_{2,1}\rangle$ of state s_2. The quotient distribution of $\mu_{2,1}$ modulo $\mathcal{R}$ is given by

$$\mu_{2,1}^{\mathcal{R}}(C) = [\mu_{2,1}]_{\mathcal{R}}(C) = \begin{cases} 3/5 & \text{if } C = C_0^{\mathcal{R}} \\ 2/5 & \text{if } C = C_1^{\mathcal{R}}. \end{cases}$$

Definition 33 (Reward and Distribution Equivalence)**.** Let $\mu_1, \mu_2 \in \mathit{Dist}(S)$ be distributions over S, $\rho_1, \rho_2 \colon S \to \mathbb{R}_{\geq 0}$ be reward functions and $\mathcal{R} \subseteq S \times S$ an equivalence relation.

(i) μ_1 and μ_2 are $\mathcal{R}$-equivalent, written $\mu_1 \equiv_{\mathcal{R}} \mu_2$, if $[\mu_1]_{\mathcal{R}} = [\mu_2]_{\mathcal{R}}$.

(ii) $\langle \mu_1, \rho_1\rangle$ and $\langle \mu_2, \rho_2\rangle$ are $\mathcal{R}$-equivalent, denoted by $\langle \mu_1, \rho_1\rangle \equiv_{\mathcal{R}} \langle \mu_2, \rho_2\rangle$, if $\mu_1 \equiv_{\mathcal{R}} \mu_2$ and $[\rho_1, \mu_1]_{\mathcal{R}} = [\rho_2, \mu_2]_{\mathcal{R}}$.

Given an equivalence relation $\mathcal{R}$ on S, distributions are equivalent under $\mathcal{R}$ if their quotient distributions under $\mathcal{R}$ coincide. This is similar for reward functions, only that they always have to be considered in the context of the accompanying distribution.

Example 34. Consider the choice $\langle \alpha, \mu_3\rangle$ of state s_3 in $\mathcal{M}$. As

$$\mu_3^{\mathcal{R}}(C) = [\mu_3]_{\mathcal{R}}(C) = \begin{cases} 3/5 & \text{if } C = C_0^{\mathcal{R}} \\ 2/5 & \text{if } C = C_1^{\mathcal{R}}, \end{cases}$$

we have $\mu_{2,1} \equiv_{\mathcal{R}} \mu_3$.

Finally, we formally phrase what it means for two states to behave equivalently.

Definition 34 (Strong Bisimulation for MRA)**.** An equivalence relation $\mathcal{R} \subseteq S \times S$ is called a *strong bisimulation* for an MRA $\mathcal{M} = \left\langle S, S^0, Act, \Delta, E, r, AP, L \right\rangle$, if for all $\langle s, t\rangle \in \mathcal{R}$, $\alpha \in Act^{\chi}$, $\mu \in Dist(S)$

(i) $L(s) = L(t)$,

(ii) $r(s) = r(t)$, and

(iii) $s \xrightarrow{\alpha}_{\rho_s} \mu \implies \exists \langle \mu', \rho_t \rangle \,.\, t \xrightarrow{\alpha}_{\rho_t} \mu' \wedge \langle \mu, \rho_s \rangle \equiv_{\mathcal{R}} \langle \mu', \rho_t \rangle$

Two states $s, t \in S$ are *strongly bisimilar*, denoted $s \sim_{\mathcal{M}} t$, if there exists a bisimulation $\mathcal{R}$ for $\mathcal{M}$ with $\langle s, t\rangle \in \mathcal{R}$. Two MRA $\mathcal{M}_i = \left\langle S_i, S_i^0, Act_i, \Delta_i, E_i, r_i, AP_i, L_i \right\rangle$ for $i \in \{1, 2\}$ are strongly bisimilar, denoted $\mathcal{M}_1 \sim \mathcal{M}_2$, if

$$\forall s_1^0 \in S_1^0 \,.\, \exists s_2^0 \in S_2^0 \,.\, s_1^0 \sim_{\mathcal{M}} s_2^0$$
$$\forall s_2^0 \in S_2^0 \,.\, \exists s_1^0 \in S_1^0 \,.\, s_1^0 \sim_{\mathcal{M}} s_2^0$$

where

$$\mathcal{M} = \left\langle S_1 \uplus S_2, S_1^0 \uplus S_2^0, Act_1 \cup Act_2, \Delta_1 \uplus \Delta_2, E, r, AP_1 \cup AP_2, L \right\rangle$$

with

$$f(s) = \begin{cases} f_1(s) & \text{if } s \in S_1 \\ f_2(s) & \text{if } s \in S_2 \end{cases}$$

for $f \in \{E, r, L\}$ is the disjoint union $\mathcal{M}_1 \uplus \mathcal{M}_2$ of the two MRA $\mathcal{M}_1$ and $\mathcal{M}_2$.

The intuition of bisimulation is that is relates states that are indistinguishable. For this, they clearly have to agree on the label as otherwise a simple atomic proposition could tell the states apart. Similarly, they need to have the same state reward. Furthermore, they are required to be able to copy each others steps in the sense that if one of them can perform a distribution μ labeled with α, then the other state can perform a step with the same action and a distribution that coincides with μ when lifted to the equivalence classes of the bisimulation. In other words, the probabilities that the two distributions assign to the equivalence classes of $\mathcal{R}$ need to match. For Markovian choices, which are labeled with $\chi(\lambda)$ for some $\lambda \in \mathbb{R}_{>0}$, requirement (iii) also ensures that s and t agree on their exit rates. In the rewarded case, additionally the expected reward value obtained in one step needs to be the same for the mimicking choice. For an MRA $\mathcal{M}$, $\sim_{\mathcal{M}}$ is the coarsest bisimulation, i. e. relates the most states and has the fewest equivalence classes among all (strong) bisimulations. If $\mathcal{M}$ is clear from the context, we omit the subscript and just write $\sim$ for $\sim_{\mathcal{M}}$.

Example 35. For our example MA $\mathcal{M}$ from Example 32 the coarsest (strong) bisimulation $\sim$ has the equivalence classes

$$S/\sim \; = \left\{ \underbrace{\{s_0\}}_{C_0}, \underbrace{\{s_1\}}_{C_1}, \underbrace{\{s_2, s_3\}}_{C_2} \right\}.$$

Conditions (i) and (ii) of Definition 34 are trivially fulfilled and (iii) can be easily verified. While the behavior of s_2 and s_3 is indistinguishable, s_0 clearly has a different behavior as it does not possess a distribution that assigns a non-zero probability to any Markovian state. As Markovian states on the one hand and probabilistic states on the other hand can by definition never be related, this implies $s_1 \not\sim s_2, s_3$.

Note that from the general definition for Markov reward automata, the definitions for discrete and continuous-time Markov chains as well as for probabilistic automata follow directly as all these models can be seen as special cases of Markov automata (see Section 2.2). In particular, Definition 34 coincides with the notion of strong bisimulation for PA as defined in [SL95], but is strictly finer than the strong bisimulation (for PA) as defined in [Seg95] by the same author. The latter allows for an α-labeled choice from state s to be matched by a *combined choices* at state t, which essentially corresponds to a weighted combination of choices labeled with α. Since our goal is the symbolic computation of $\sim_{\mathcal{M}}$ and — to the best of our knowledge — all algorithms to compute this weaker formulation of bisimulation need to repeatedly solve large linear programs, we do not treat this refined version of strong bisimulation further.

Given a bisimulation $\mathcal{R}$ for an MA $\mathcal{M}$, we can now define the quotient model.

Definition 35 (Bisimulation Quotient)**.** Let $\mathcal{R}$ be a strong bisimulation for an MRA $\mathcal{M} = \langle S, S^0, Act, \Delta, E, r, AP, L\rangle$. The *quotient Markov reward automaton* is defined by

$$\mathcal{M}/\mathcal{R} = \langle S/\mathcal{R}, S^0/\mathcal{R}, Act, \Delta/\mathcal{R}, E/\mathcal{R}, r/\mathcal{R}, AP, L/\mathcal{R}\rangle$$

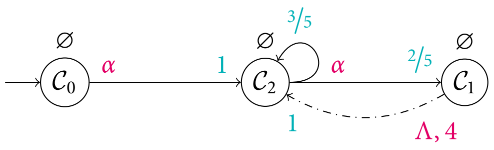


Figure 5.2: The quotient of the example MRA $\mathcal{M}$ with respect to $\sim_{\mathcal{M}}$.

where

» $L/\mathcal{R}([s]_{\mathcal{R}}) = L(s)$,

» $r/\mathcal{R}([s]_{\mathcal{R}}) = r(s)$,

» $E/\mathcal{R}([s]_{\mathcal{R}}) = E(s)$, and

» $\Delta/\mathcal{R}$ is the (unique) smallest relation satisfying

$$\frac{\langle s, \alpha, \rho, \mu\rangle \in \Delta}{\langle [s]_{\mathcal{R}}, \alpha, [\rho, \mu]_{\mathcal{R}}, [\mu]_{\mathcal{R}}\rangle \in \Delta/\mathcal{R}.}$$

The quotient Markov reward automaton has as many states as there are equivalence classes in $\mathcal{R}$. The behavior of an equivalence class $[s]_{\mathcal{R}}$ corresponds to lifting the distributions of the choices of s to the quotient state space. Note that the labeling, state reward and exit rate functions are well-defined, because Definition 34 requires that states in the same equivalence class agree on these values.

Example 36. Recall the bisimulation $\sim$ with the equivalence classes $\mathcal{C}_0, \mathcal{C}_1$ and $\mathcal{C}_2$ from Example 35. The quotient MA $\mathcal{M}/\sim$ is depicted in Figure 5.2.

Clearly, the quotient MRA is at most as large as the original MRA (in terms of states). In general, it may be smaller, as states are potentially merged. However, as it lumped only states that were intuitively indistinguishable, it preserves interesting properties. We will now make the relation of the quotient and the original model more precise and show what this means in terms of the preservation of logical properties.

Lemma 1. *Let $\mathcal{M}$ be an MRA and $\mathcal{R}$ be a strong bisimulation for $\mathcal{M}$. Then $\mathcal{M} \sim \mathcal{M}/\mathcal{R}$.*

Lemma 1 states that with respect to strong bisimulation, $\mathcal{M}$ and $\mathcal{M}/\mathcal{R}$ are indistinguishable. Finally, we want to discuss which logical properties are preserved by quotienting with respect to bisimulation. The following theorem summarizes the results regarding the logical characterization.

Theorem 1 (Bisimulation Equivalence and CSL/PCTL*). *Let $\mathcal{M}$ be an MRA and $\mathcal{R}$ be a strong bisimulation for $\mathcal{M}$.*

(i) $\langle s, t\rangle \in \mathcal{R} \implies s \equiv_{\mathrm{PCTL}^*} t,$

(ii) $\langle s, t\rangle \in \mathcal{R} \implies s \equiv_{\mathrm{CSL}} t.$

That is, bisimilar states are not distinguishable by neither PCTL* [SZG13] nor CSL. The latter result was proven by Sergey Sazonov in his Master's thesis [Saz13], but is not published to this date. The slightly weaker result that strongly bisimilar states of an interactive Markov chain (IMC) are not distinguished by CSL is shown in [NK07]. To the best of our knowledge, there is no result that formally establishes preservation of (expected) reward objectives. However, in the same spirit as [Tim13], we conjecture that (strong) bisimulation on MRA preserves all these properties.

Corollary 1. *Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two MRA. Then*

(i) $\mathcal{M}_1 \sim \mathcal{M}_2 \implies \mathcal{M}_1 \equiv_{\mathrm{PCTL}^*} \mathcal{M}_2$, *and*

(ii) $\mathcal{M}_1 \sim \mathcal{M}_2 \implies \mathcal{M}_1 \equiv_{\mathrm{CSL}} \mathcal{M}_2$.

Theorem 1 and Corollary 1 lay the foundation of the bisimulation minimization approach. Instead of verifying a property φ on the given MRA $\mathcal{M}$, a bisimulation relation $\mathcal{R}$ for $\mathcal{M}$ is computed. Then, the quotient model $\mathcal{M}/\mathcal{R}$ is extracted. Because of the strong logic preservation properties, the formula may now be verified on the quotient and the result carries over directly to the original model. Note that the minimal quotient size is obtained by choosing $\mathcal{R} = \sim_{\mathcal{M}}$. Sometimes it might be computationally easier to compute a finer bisimulation at the expense of a larger quotient state space [KNP06b; DM06]. However, in the context of this thesis, we aim for computing $\sim_{\mathcal{M}}$.

5.3 Partition Refinement

We are now turning to the algorithmic computation of $\sim$. For the rest of this chapter, we fix a finite MRA $\mathcal{M} = \langle S, S^0, Act, \Delta, E, r, AP, L\rangle$. We assume that $\mathcal{M}$ has been subject to the maximal progress cut (see Section 2.2). Suppose that Π is a partition of the state space S that is strictly coarser than $\sim$, i. e., $\sim \sqsubset \Pi$. Then it is straightforward to say when two states *cannot* be bisimilar, by looking at their one-step behavior with respect to Π. More concretely, assume there are two states $s, t \in B$ within a block $B \in \Pi$, and s can make a step $s \xrightarrow{\alpha}_{\rho_s} \mu$ that cannot be matched by an equivalent choice of state t in the sense of Definition 34 requirement (iii). In this case, we call the block B *unstable* and we clearly have $s \not\sim t$. This justifies the following approach that is typically referred to as *partition refinement*. Starting with an initial partition Π_0, the algorithm continuously refines the partition by checking whether there are blocks that are unstable with respect to the requirements of Definition 34. If so, these blocks are split into subblocks such that within each of these subblocks only states are grouped that could potentially be bisimilar. Starting from an initial partition Π_0 that respects the criteria (i) and (ii) of Definition 34, this process yields a bisimulation relation. A straightforward choice for the initial partition is

$$\Pi_0^{\mathcal{M}} = \{\{t \in S \mid L(t) = L(s) \wedge r(t) = r(s) \wedge E(t) = E(s)\} \mid s \in S\}.$$

Even though the requirements on the exit rates is not explicit in Definition 34, it is implied by requirement (iii).

Example 37. For the MRA $\mathcal{M}$ in Figure 5.1, the initial partition $\Pi_0^{\mathcal{M}}$ is given by

$$\Pi_0^{\mathcal{M}} = \left\{ \underbrace{\{s_0, s_2, s_3\}}_{\mathcal{C}_0^{\mathcal{R}}}, \underbrace{\{s_1\}}_{\mathcal{C}_1^{\mathcal{R}}} \right\} = S/\mathcal{R}$$

where $\mathcal{R}$ is the equivalence relation of Example 33. Although the labeling $\varnothing$ is the same for all the states, the exit rates of Markovian and probabilistic states always differ and the initial partition separates those states. Block $\mathcal{C}_0^{\mathcal{R}}$ is unstable: for the distribution μ_0 available in s_0 we have $[\mu_0]_{\mathcal{R}} = \delta_{\mathcal{C}_0}$ that cannot be matched by neither s_2 nor s_3 as all their quotient distributions assign a non-zero probability to $\mathcal{C}_1$. Splitting $\mathcal{C}_0$ according to requirement (iii) of Definition 34 yields the partition $S/\sim$.

The partition refinement procedure can be elegantly phrased in terms of signatures of states and signature-based refinement. Intuitively, a signature is a function mapping states to a characterization of their behavior with respect to a partition Π. In the case of strong bisimulation for MRA, we define the signature as follows.

Definition 36 (Signature of a State in an MRA)**.** Let $s \in S$ and Π be a partition of S. The signature of s with respect to Π is given by

$$\mathrm{sig}_\Pi(s) = \left\{ \left\langle \chi^{-1}(\alpha), [\rho, \mu]_\Pi, [\mu]_\Pi \right\rangle \mid s \xrightarrow{\alpha}_\rho \mu \right\}.$$

Hence, the signature of a state s captures the one-step behavior of s lifted with respect to the partition Π. Note that for continuous-time models (CTMCs and MA), the signature abstracts from the exit rate of Markovian states. Intuitively, we can do this because we move this criterion to the initial partition $\Pi_0^{\mathcal{M}}$ by requiring related states to agree on their exit rates. For models without transition rewards, we omit the (quotient) reward functions from the signature altogether. For our definitions of Markov chains (i) all distributions are labelled equally (with τ for DTMCs and Λ for CTMCs), (ii) there are no rewards, and (iii) the signature of a state is a singleton. We therefore simplify the notation for such models by leting the signature of a state $s \in S$ be directly given by the (unique) quotient distribution $[\mathbf{P}(s)]_\Pi$ of s.

Example 38. Reconsider the quotient distributions $\mu_{2,1}^{\mathcal{R}} = \mu_3^{\mathcal{R}}$ of states s_2 and s_3 from Examples 33 and 34 and let $\Pi = \Pi_0^{\mathcal{M}}$ as in Example 37. Then

$$\mathrm{sig}_\Pi(s_2) = \left\{ \left\langle \alpha, \mu_{2,1}^{\mathcal{R}} \right\rangle \right\}$$

is the signature of s_2 with respect to Π. Note that it only has one element, because the quotient distribution of both of s_2's choices coincide. As it turns out, $\mathrm{sig}_\Pi(s_2) = \mathrm{sig}_\Pi(s_3)$. However, we find the signature of s_0 to be

$$\mathrm{sig}_\Pi(s_0) = \{ \langle \alpha, \delta_{C_0} \rangle \}$$

and as $\delta_{C_0} \neq \mu_{2,1}^{\mathcal{R}}$, it is $\mathrm{sig}_\Pi(s_0) \neq \mathrm{sig}_\Pi(s_2) = \mathrm{sig}_\Pi(s_3)$. The signature of the Markovian state s_1 is

$$\mathrm{sig}_\Pi(s_1) = \{ \langle \Lambda, \delta_{C_0} \rangle \}.$$

We can now define an operator that takes a partition Π as input and returns a refined partition $\Pi' \sqsubseteq \Pi$ based on the signatures sig_Π.

Definition 37 (Sigref Operator)**.** The signature-refinement (sigref) operator for partition Π is defined as

$$\text{sigref}(\Pi) = \Big\{\big\{t \in S \mid \text{sig}_\Pi(t) = \text{sig}_\Pi(s) \land t \equiv_\Pi s\big\} \mid s \in S\Big\}.$$

The sigref operator groups all states into one block that have an identical signature *and* have been related before. The latter condition ensures that the operator is monotonic in the presence of non-monotonic signatures that map states to the same signature even though the states are already not equivalent in the previous partition. For example, it is quite common that the initial partition Π_0 separates states that have a similar behavior, but do not possess the same state labeling (or state rewards). The signatures of these states are then potentially equal, but they are easily distinguished by an atomic proposition. Without the requirement that related states also had to be related in the input partition, the sigref operator would group those states in the same block, even if they are not bisimilar.

Example 39. As we have seen in Example 38, the signatures of s_2 and s_3 agree with respect to $\Pi_0^{\mathcal{M}}$, so

$$\text{sigref}(\Pi_0^{\mathcal{M}}) = \left\{ \underbrace{\{s_0\}}_{C_0}, \underbrace{\{s_1\}}_{C_1}, \underbrace{\{s_2, s_3\}}_{C_2} \right\} = S/\sim$$

It is easy to verify that $\text{sigref}(S/\sim) = S/\sim$.

The sigref operator satisfies the following properties.

Lemma 2 (Correctness)**.** *The following two statements hold.*

(i) *If* $\sim \sqsubseteq \Pi$, *then* $\sim \sqsubseteq \text{sigref}(\Pi)$

(ii) $\Pi \sqsupseteq \Pi_0^{\mathcal{M}}$ *and* $\text{sigref}(\Pi) = \Pi$ *if and only if* Π *is a strong bisimulation for* $\mathcal{M}$.

First, starting with a partition Π that is coarser than $\sim$, the result of the application of sigref to Π using the signature sig_Π is still coarser than $\sim$. In other words, $\text{sigref}(\Pi)$ does not separate states that are in fact bisimilar. Secondly, a partition is a bisimulation

Algorithm 1: Signature-based partition refinement.

```
1 function SignatureRefinement($\mathcal{M}$, $\Pi_0$):
      input: $\mathcal{M}$: the MRA for which to compute $\sim_{\mathcal{M}}$,
             $\Pi_0$: the initial partition
      output: the partition $\sim_{\mathcal{M}}$
2     $i \leftarrow 0$
3     repeat
4         $i \leftarrow i + 1$
5         $\Pi_i = \mathrm{sigref}(\Pi_{i-1})$
6     until $\Pi_i = \Pi_{i-1}$
7     return $\Pi_i$
```

if and only if the partition is a refinement of $\Pi_0^{\mathcal{M}}$ and it is a fixed-point of the sigref operator. In this case all blocks are stable with respect to the current partition.

With the sigref operator in place, the partition-refinement algorithm can be written as in Algorithm 1. Ultimately, it computes a sequence of partitions

$$\Pi_0 \sqsupseteq \Pi_1 \sqsupseteq \dots \qquad \text{with} \qquad \Pi_{i+1} = \mathrm{sigref}(\Pi_i).$$

Since the sigref operator is monotonic and the set of partitions of S that refine $\Pi_0^{\mathcal{M}}$ together with the $\sqsubseteq$ relation is a (finite) complete lattice, this sequence eventually stabilizes in the sense that there exists an index n such that $\Pi_n = \Pi_{n+1}$. At this point, the algorithm can terminate as no further changes to the partitions will occur. By the Kleene fixpoint theorem, the partition Π_n is the greatest fixed point of sigref that refines $\Pi_0^{\mathcal{M}}$ and therefore equal to $\sim$. Using the properties of the sigref operator stated in Lemma 2, the following corollary states the correctness of Algorithm 1.

Corollary 2. *SignatureRefinement*$(\mathcal{M}, \Pi_0^{\mathcal{M}}) = \sim$.

5.4 MTBDD-based Signature Refinement

In this section, we describe how signature-based partition refinement can be implemented in a purely symbolic fashion. That is, given an MRA stored in terms of DDs, the procedure SignatureRefinement is performed exclusively at the decision diagram level. For this, recall the encoding of an MRA $\mathcal{M} = \langle S, S^0, Act, \Delta, E, r, AP, L \rangle$ in terms of DDs in Section 2.5 on page 36. More specifically, we encoded $\mathcal{M}$ by means of the tuple $\left\langle \mathsf{B}_S, \mathsf{B}_{S^0}, \mathsf{M}_\Delta, \mathsf{M}_\Delta^\rho, \mathsf{M}_E, \mathsf{M}_r, (\mathsf{B}_a)_{a \in AP} \right\rangle$ over the variables

$$Var_{\mathcal{M}} = \underbrace{\{x_1, \ldots, x_n\}}_{\mathcal{S}} \uplus \underbrace{\{x_1', \ldots, x_n'\}}_{\mathcal{S}'} \uplus \underbrace{\{\mathfrak{a}_1, \ldots, \mathfrak{a}_k\}}_{\mathcal{A}} \uplus \underbrace{\{\mathfrak{n}_1, \ldots, \mathfrak{n}_\ell\}}_{\mathcal{N}}$$

of DDs consisting of

» a BDD B_S over $\mathcal{S}$ encoding S,

» a BDD B_{S^0} over $\mathcal{S}$ encoding S^0,

» the BDDs B_a over $\mathcal{S}$ encoding whether $a \in L(s)$ for each $a \in AP$,

» an MTBDD M_Δ over $Var_{\mathcal{M}}$ encoding the probability distributions available in the states according to Δ without the transition rewards,

» an MTBDD M_Δ^ρ over $Var_{\mathcal{M}}$ encoding the transition rewards of all distributions according to Δ,

» an MTBDD M_E over $\mathcal{S}$ encoding the exit rate function E,

» an MTBDD M_r over $\mathcal{S}$ encoding the state rewards r.

To implement SignatureRefinement symbolically, we discuss how to (i) represent a partition Π of S, (ii) compute $\Pi_0^{\mathcal{M}}$, (iii) compute and represent sig_Π, and (iv) implement the sigref operator — all in terms of DDs.

Example 40. Reconsider the MA $\mathcal{M} = \left\langle S, S^0, Act, \Delta, E, AP, L \right\rangle$ from Example 32. For improved readability, we assume the variable order

$$\underbrace{\mathfrak{s}_1 < \mathfrak{s}_0}_{\mathcal{S}} < \underbrace{\mathfrak{n}_0}_{\mathcal{N}} < \underbrace{\mathfrak{a}_0}_{\mathcal{A}} < \underbrace{\mathfrak{s}_1' < \mathfrak{s}_0'}_{\mathcal{S}'} .$$

and give the DD representation of Δ in Figure 5.3. Here, we encode every state s_i over the variables $\mathcal{S}$ and $\mathcal{S}'$, respectively, using the standard binary encoding of i. For example,

$$(\mathcal{S} \leftarrow \langle s_2 \rangle)(\mathfrak{s}) = \begin{cases} 0 & \text{if } \mathfrak{s} = \mathfrak{s}_0 \\ 1 & \text{if } \mathfrak{s} = \mathfrak{s}_1 \end{cases}$$

encodes the state s_2. Note that $\mathfrak{s}_1$ contains the most significant bit. Furthermore, we

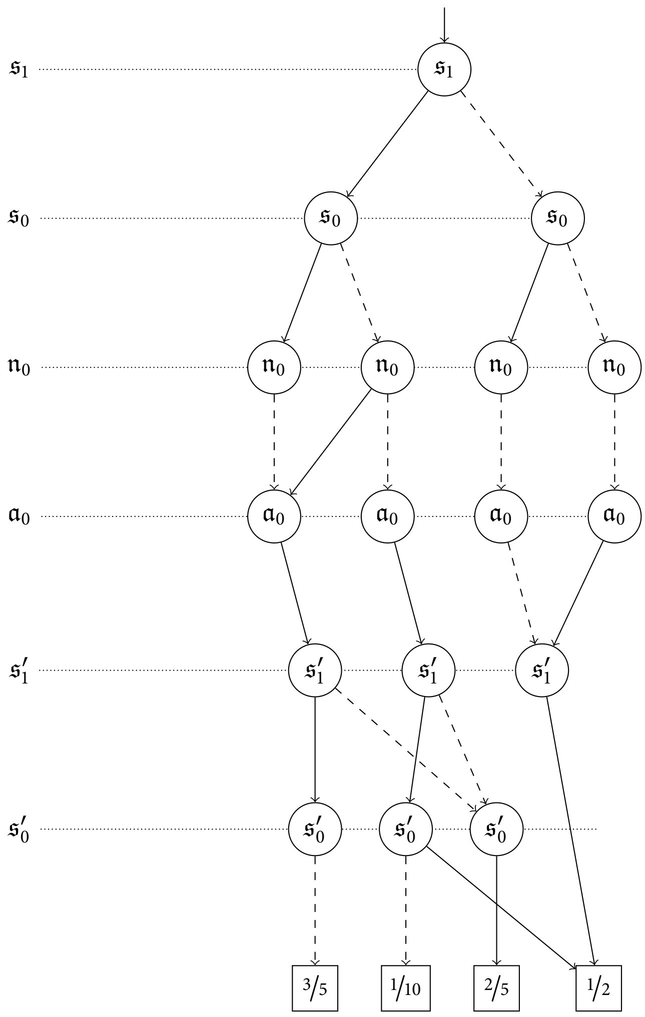


Figure 5.3: The MTBDD M_Δ for $\mathcal{M}$.

use

$$(\mathcal{A} \leftarrow \langle \alpha' \rangle)(\mathfrak{a}_0) = \begin{cases} 0 & \text{if } \alpha' = \Lambda \\ 1 & \text{if } \alpha' = \alpha \end{cases}$$

and use the single variable $\mathfrak{n}_0$ to disambiguate the two distributions of s_2.

5.4.1 Representation of Partitions

Let $\Pi = \{B_0, \ldots, B_{h-1}\}$ be a partition of the state space S of $\mathcal{M}$ with h blocks. As discussed in [Wim10], there are various ways to represent this partition. In the context of the sigref operator, however, one representation turns out to be the most appealing. For this encoding, we introduce additional *block variables* $\mathcal{B} = \{b_1, \ldots, b_{n+k+\ell}\}$ that are used to encode the block indices. For the implementation of the sigref operator, it is important that the block variables are all greater than the state variables $\mathcal{S}$ in the variable ordering. More concretely, we require $\mathcal{S}, \mathcal{N} < \mathcal{A} < \mathcal{B}$. Other than that, the variables can be ordered arbitrarily. For example, the nondeterminism variables $\mathcal{N}$ may be on top, below or interleaved with the state variables. We then encode Π as the BDD

$$\mathsf{B}_\Pi(\mathcal{S} \leftarrow \langle s \rangle, \mathcal{B} \leftarrow \langle \kappa \rangle) = \begin{cases} 1 & \text{if } s \in B_\kappa \\ 0 & \text{otherwise} \end{cases}$$

Note that for this encoding to be possible, we would only need n Boolean variables in $\mathcal{B}$. In the context of nondeterministic models (see Section 5.4.4 on 138), it will become apparent later how the additional $k + \ell$ variables are used.

Example 41. Figure 5.4 shows the BDD representing the partition $\Pi_0^{\mathcal{M}}$ from Example 37. It uses the block variables

$$\underbrace{b_0 < b_1}_{\mathcal{B}}$$

that are ordered below all other variables in Var. As before, the variable b_1 is used to represent the most significant bit of the block indices encoded in terms of $\mathcal{B}$, but unlike the state variables, we choose to let the lower bit precede it. The nodes p_0 and p_1 therefore encode the blocks $\mathcal{C}_0$ and $\mathcal{C}_1$, respectively.

Representing a partition like this has several advantages. First of all, it requires only a single BDD to represent the whole partition, regardless of the number of blocks in the

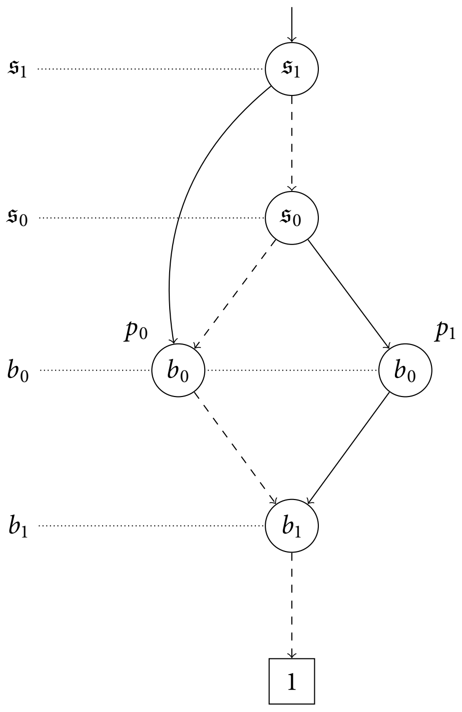


Figure 5.4: The BDD representation of the initial partition $\Pi_0^{\mathcal{M}}$.

partition. This greatly simplifies the symbolic implementation of the sigref operator. While it is true that the size of B_Π grows linearly in the number of blocks, experimental evaluation has shown that it is more compact [Wim10] than, for example, the relational representation in terms of a BDD for

$$\mathsf{B}_R(\mathcal{S} \leftarrow \langle s \rangle, \mathcal{S}' \leftarrow \langle s' \rangle) = 1 \iff s \equiv_\Pi s'.$$

For reasons that will become apparent later, we defer explaining how the initial partition $\Pi_0^{\mathcal{M}}$ can be computed to Section 5.4.3 until after the presentation of the symbolic implementation of the sigref operator for MCs.

5.4.2 Symbolic Implementation of the sigref Operator for MCs

Similar to our presentation of encoding probabilistic models in terms of DDs, we treat the simpler case of MCs as a precursor. Therefore, suppose that our model $\mathcal{M}$ is actually a DTMC $\mathcal{M} = \mathcal{D} = \langle S, s^0, \mathbf{P}, AP, L \rangle$.

Example 42. For illustration purposes, we consider the DTMC $\mathcal{D} = \langle S, s^0, \mathbf{P}, AP, L \rangle$ that results from $\mathcal{M}$ (see Example 32) by

» relabeling all actions to τ (and therefore treating s_1 as probabilistic),

» deleting the choice of $\mu_{2,1}$ at state s_2, and

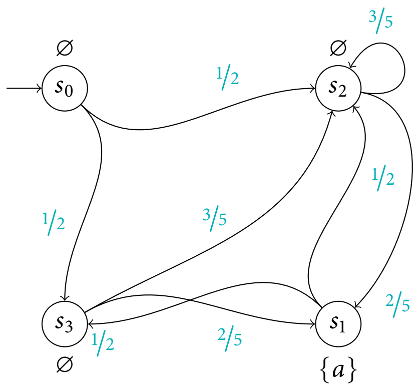
» attaching the label $\{a\}$ to s_1 instead of $\varnothing$.

This DTMC is shown in Figure 5.5(a). The new label attached to s_1 has the sole purpose of having $\Pi_0^{\mathcal{D}} = \Pi_0^{\mathcal{M}}$ in the absence of Markovian states. Figure 5.5(b) shows the transition relation MTBDD $\mathsf{M}_{\mathbf{P}}$ over the variables $\mathcal{S}$ and $\mathcal{S}'$.

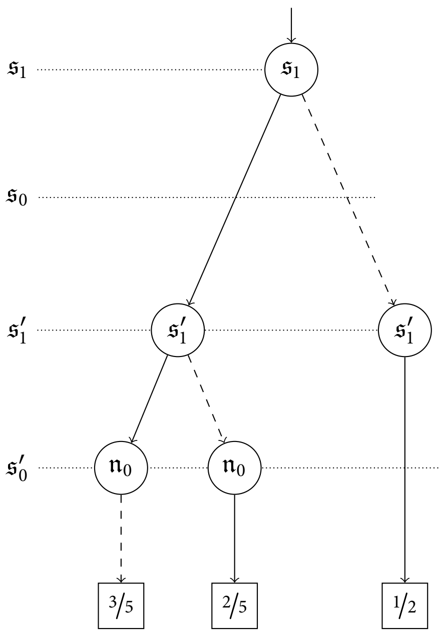
We implement the sigref operator in terms of a two-step procedure as in [DP18]. First, an MTBDD $\mathsf{M}_{\text{sig}}^{\Pi}$ is computed that represents the signatures of all states in $\mathcal{D}$ with respect to the current partition Π. Then, a traversal of $\mathsf{M}_{\text{sig}}^{\Pi}$ suffices to generate the partition $\Pi' = \text{sigref}(\Pi)$. Note that this approach refines all blocks of the partition simultaneously and is therefore somewhat different from the variants shown in [Wim10]. The latter either (i) refine only one block in one iteration, (ii) only refine the blocks with the same state labeling in one iteration, or (iii) require additional MTBDD variables and changes to the signature definition and computation. The alternative approach suggested in [DP18] not only has the advantage that it refines all blocks simultaneously without invasive changes, but at the same time it partially reuses block encodings over successive applications of the sigref operator. This may not appear to be a substantial change, but because of the implementation details of DDs this can improve the runtimes by several orders of magnitude [DP18].

Let us turn to the computation of the MTBDD $\mathsf{M}_{\text{sig}}^{\Pi}$ for the signatures based on the BDD B_{Π} for a given partition Π. Because of the choice of the partition representation, we can compute it as

$$\mathsf{M}_{\text{sig}}^{\Pi} = \textsc{SumAbstract}\left(\mathcal{S}', \mathsf{M}_{\mathbf{P}} \cdot \textsc{Rename}\left(\mathsf{B}_{\Pi}, \mathcal{S}, \mathcal{S}'\right)\right). \tag{5.1}$$



(a) The graph of $\mathcal{D}$.



(b) The transition MTBDD $\mathsf{M_P}$ for the DTMC $\mathcal{D}$.

Figure 5.5: The DTMC "submodel" $\mathcal{D}$ of the MA $\mathcal{M}$.

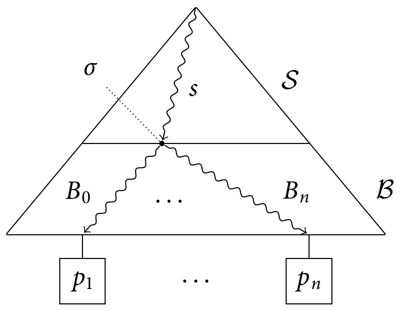


Figure 5.6: The shape of the signature MTBDD for DTMCs.

By construction, we have

$$\mathsf{M}^{\Pi}_{\mathrm{sig}}(\mathcal{S} \leftarrow \langle s \rangle, \mathcal{B} \leftarrow \langle \kappa \rangle) = \sum_{s' \in B_\kappa} \mathbf{P}(s)(s') = [\mathbf{P}(s)]_\Pi(B_\kappa).$$

and, consequently, it is a representation of the signature for DTMCs. Given our variable order, $\mathsf{M}^{\Pi}_{\mathrm{sig}}$ has a property that we can exploit in the implementation of the refinement. Figure 5.6 sketches the shape of the signature MTBDD. It is built over the variables $\mathcal{S} \uplus \mathcal{B}$ as state-block pairs $\langle s, B \rangle$ are mapped to the quotient probability with which s can reach B in one step. As $\mathcal{S} \prec \mathcal{B}$, the state encodings strictly precede the block encodings. Therefore, for a fixed $s \in S$, the MTBDD $\mathsf{M}_s = \mathsf{M}^{\Pi}_{\mathrm{sig}}|_{\mathcal{S} \leftarrow \langle s \rangle}$ is a representation of the quotient distribution $\mu = [\mathbf{P}(s)]_\Pi$ of s with respect to the partition Π. Also, because of the variable ordering, the node representing μ is a subnode of $\mathsf{M}^{\Pi}_{\mathrm{sig}}$ as the cofactor can be computed by simply following the state encoding. Since we consider only fully reduced MTBDDs, there is exactly one node in $\mathsf{M}^{\Pi}_{\mathrm{sig}}$ that represents $[\mathbf{P}(s)]_\Pi$. This holds true for any arbitrary $s \in S$, and we make the following observation: $s, s' \in S$ have the same signature if and only if following the two state encodings $\langle s \rangle$ and $\langle s' \rangle$ in $\mathsf{M}^{\Pi}_{\mathrm{sig}}$ results in the very same DD node.

Example 43. Reconsider the DTMC $\mathcal{D}$ and the MTBDD $\mathsf{M}_{\mathbf{P}}$ representation of its transition function $\mathbf{P}$ from Example 42. The signature MTBDD $\mathsf{M}^{\Pi}_{\mathrm{sig}}$ with respect to $\Pi = \Pi^{\mathcal{D}}_0 = \Pi^{\mathcal{M}}_0$ (see Example 37 and Example 41 for its BDD representation) is depicted in Figure 5.7. The node σ_0 represents the quotient distribution δ_{C_0}. Similarly, the node σ_1 represents the quotient distribution $\mu^{\mathcal{R}}_{2,1}$ of Example 33.

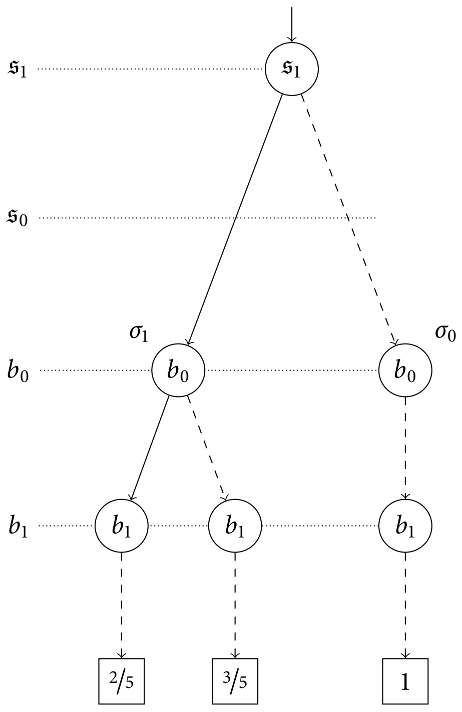


Figure 5.7: The signature MTBDD $\mathsf{M}_{\text{sig}}^{\Pi}$ for the DTMC $\mathcal{D}$.

This observation gave rise to the core idea of the algorithms proposed in [Wim10]. Starting from the root node, $\mathsf{M}_{\text{sig}}^{\Pi}$ is traversed in a depth-first fashion. The search descends as long as the currently visited node is labeled with a variable in $\mathcal{S}$. As soon as this is not the case anymore, the search has hit a node σ that represents the signature of a state. If this node has been seen before, the current state is placed into the same block as the previous ones whose encodings led to σ. In case the node has not been seen before, the current state is assigned to a new block. Proceeding this way, all blocks are grouped according to their signatures by a simple traversal of the signature MTBDD.

In [DP18], the authors propose to not only traverse the signature MTBDD, but *simultaneously* traverse the current partition BDD. A slightly simplified version of the resulting procedure is shown in Algorithm 2. Specifically, as multi-threading is one of the main goals of [DP18], but is not of major concern here, we simplify the corresponding parts to a sequential setting.

Given the root node σ of the signature MTBDD $\mathsf{M}_{\text{sig}}^{\Pi}$, the root node p of the partition BDD B_{Π} and a set of *state variables* $\mathit{Var}' \subseteq \mathit{Var}$, the algorithm follows the approach described above. It maintains an (initially empty) mapping *cache* to store the signatures of blocks that have already been discovered. Furthermore, it takes as input a counter variable that is used to determine the next free block encoding. First, lines 2 and 3 make sure that illegal state encodings are not mapped to any block by checking whether the previous partition does not assign the current state encoding to any block. Line 4 checks whether the pair of nodes $\langle \sigma, p \rangle$ has already been seen in the current traversal. If so, the cached result is returned. In the simple traversal of [Wim10], the cached result only depends on the signature node σ. For the dual (signature and partition) traversal, however, the result may depend on the partition node p. Intuitively, this is because states with the same signature must not be grouped together if the node in B_{Π} representing their block in Π are different, as this means they are not related in the current partition Π. The lines 7-10 recursively descend if the next relevant variable in either the signature or the partition is a state variable. Finally, lines 11-16 assign the current state encoding to a new block. Note that "new" is to be understood with respect to the current top-level invocation of the Refine. The block may be a block encoding that can be reused from the old partition (lines 12-14) or a completely new block (line 16). In the latter case, EncodeBlock(b) is assumed to encode the index b in terms of the variables $\mathcal{B}$ and increment the counter b such that it is visible to the outside.

The simultaneous traversal of both $\mathsf{M}_{\text{sig}_{\Pi}}$ and B_{Π} has two major advantages. First, it makes the monotonicity requirement of the sigref operator trivial to implement. Recall that the operator must only relate states that have been related before. By also considering the current partition in the traversal, it is easy to keep states separate that were not related before. Second and most importantly, it allows to *reuse* the block encodings over iterations. If a signature node is hit, the old block encoding can potentially be reused. To this end, a set of blocks is maintained in the form of an (initially empty) set *used* of BDD nodes by the algorithm that keeps track of which nodes (representing block encodings) were already reused from the partition Π. While this seems like a minor optimization, we illustrate its impact on performance in Section 5.6.

Example 44. We apply Refine to the signature MTBDD $\mathsf{M}_{\text{sig}}^{\Pi}$ from Example 43. As the signature is traversed simultaneously with the BDD B_{Π}, recall that the latter is given in Figure 5.4. Starting from the root nodes of the DDs, the first non-trivial

Algorithm 2: Implementation of the sigref operator for MCs.

```
1  function Refine($\sigma, p, Var', b$):
       input: $\sigma$: root node of the signature MTBDD,
              $p$: root node of the previous partition BDD,
              $Var'$: the set of state variables,
              $b$: the next free block number
       output: the next partition BDD
2      if $p = $ Const $(0)$ then
3          return $p$
4      if $\langle \sigma, p \rangle \in cache$ then                // check if node pair already seen
5          return $cache[\langle \sigma, p \rangle]$
6      $x \leftarrow$ TopVar $(\sigma, p)$
7      if $x \in Var'$ then                                  // descend if $x$ is state variable
8          $low \leftarrow$ Refine$(\sigma|_{x=0}, p|_{x=0}, Var', cache, b)$
9          $high \leftarrow$ Refine$(\sigma|_{x=1}, p|_{x=1}, Var', cache, b)$
10         $result \leftarrow$ Ite $(x, high, low)$
11     else                                                  // hit a signature encoding
12         if $p \notin used$ then                           // reuse block $p$ if possible
13             $used \leftarrow used \cup \{p\}$
14             $result \leftarrow p$
15         else                                              // otherwise use new block number
16             $result \leftarrow$ EncodeBlock$(b)$
17     $cache[\langle \sigma, p \rangle] \leftarrow result$
18     return $result$
```

action that happens is the visit to the node pair $\langle \sigma_0, p_0 \rangle$. As σ_0 is not labeled with a state variable, it is detected as representing a signature. As p_0 encodes the block C_0, which is not yet claimed in the current traversal, this block is returned for the node pair. The next interesting visit is for the node pair $\langle \sigma_0, p_1 \rangle$. The signature node is the same as before, but the (previous) partition node is different. That means that even though the signatures of the two states (s_0 and s_1) is the same with respect to Π, we have $s_0 \not\equiv_\Pi s_1$. As the block C_1 encoded by p_1 is not yet claimed either, it is returned. Finally, the pair $\langle \sigma_1, p_0 \rangle$ is visited. The signature node σ_1 has not been visited before, but the block encoded by p_0 is already taken, so the new block with index 2 encoded

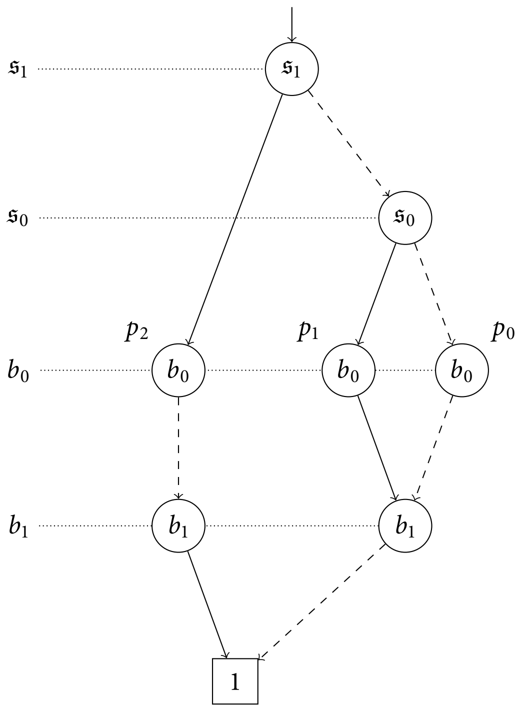


Figure 5.8: The BDD encoding the partition $S/{\sim}$.

by the node p_2 is returned. The resulting BDD, depicted in Figure 5.8, represents the partition $S/{\sim}$.

Algorithm 3 embeds the Refine algorithm into the framework of Algorithm 1. In particular, it shows how the global block counter b is maintained throughout the whole traversal process.

5.4.3 Computation of the Initial Partition

We now show how to obtain the initial partition $\Pi_0^{\mathcal{M}}$ using the representation we introduced in Section 5.4.1. For this, we take a two-step approach. The first step consists

Algorithm 3: Symbolic signature-based partition refinement for MCs.

```
1  function EncodeBlock($b$):
       input: $b$: the counter holding the next free block number
       output: the encoding of the next free block number as a BDD and an increment to
               the counter $b$ as a side-effect
2      $result \leftarrow$ Encode $(\mathcal{B} \leftarrow \langle b \rangle)$
3      $b \leftarrow b + 1$                                    // increment counter globally
4      return result
5  function SignatureRefinement($\mathcal{M}, \Pi_0$):
       input: $\mathcal{M}$: the MC for which to compute $\sim_{\mathcal{M}}$,
              $\Pi_0$: the initial partition
       output: the bisimilarity relation $\sim_{\mathcal{M}}$
6      $i \leftarrow 0$
7      $b \leftarrow |\Pi_0|$                                  // initialize counter to partition size
8      repeat
9          $i \leftarrow i + 1$
10         $\mathsf{M}^{\Pi}_{\text{sig}} \leftarrow$ SumAbstract $(\mathcal{S}', \mathsf{M}_{\mathbf{P}} \cdot$ Rename $(\mathsf{B}_{\Pi_{i-1}}, \mathcal{S}, \mathcal{S}'))$
11         $\mathsf{B}_{\Pi_i} \leftarrow$ Refine$(\mathsf{M}^{\Pi}_{\text{sig}}, \mathsf{B}_{\Pi_{i-1}}, \mathcal{S}, b)$     // refine signature MTBDD $\mathsf{M}^{\Pi}_{\text{sig}}$
12     until $\mathsf{B}_{\Pi_i} = \mathsf{B}_{\Pi_{i-1}}$
13     return $\mathsf{B}_{\Pi_i}$
```

of computing the representation of the partition

$$\Pi_0^L = \{\{t \in S \mid L(t) = L(s)\} \mid s \in S\}$$

that only respects the state labeling. In the second step, we account for the state rewards and the exit rates of the MRA. Algorithm 4 shows how Π_0^L can be computed in terms of elementary BDD operations. It maintains a set of BDDs that ultimately characterize all labeling sets that occur in $\mathcal{M}$ (lines 3-11). After these have been computed, it attaches a unique block encoding to these blocks and combines them to the BDD $\mathsf{B}_{\Pi_0^L}$ (lines 12-15).

We now consider the state rewards r and exit rates E of $\mathcal{M}$. We observe that an MTBDD M over the variables $\mathcal{S}$ can also be seen as signature MTBDDs over the variables $\mathcal{S} \uplus \mathcal{B}$ even though there is no node in M labeled with a variable in $\mathcal{B}$. In particular, this holds for both M_r and M_E. This makes refining the partition Π_0^L with respect to state rewards

Algorithm 4: The algorithm to compute Π_0^L.

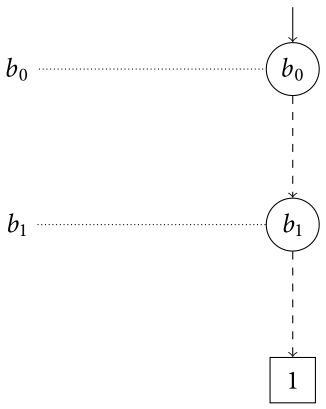
```
function ComputeLabelPartition($\mathcal{M}$):
    input: $\mathcal{M}$: the MRA for which to compute $\Pi_0^L$
    output: the partition $\Pi_0^L$ represented as a BDD $\mathsf{B}_{\Pi_0^L}$
    $B \leftarrow \{\textsc{Const}(1)\}, B' \leftarrow \varnothing$
    foreach $a \in AP$ do
        foreach $\mathsf{B} \in B$ do
            $\mathsf{B}_t \leftarrow \mathsf{B} \wedge \mathsf{B}_a$
            if $\mathsf{B}_t \neq \textsc{Const}(0)$ then
                $B' \leftarrow B' \cup \{\mathsf{B}_t\}$
            $\mathsf{B}_t \leftarrow \mathsf{B} \wedge \neg\mathsf{B}_a$
            if $\mathsf{B}_t \neq \textsc{Const}(0)$ then
                $B' \leftarrow B' \cup \{\mathsf{B}_t\}$
        $B \leftarrow B', B' \leftarrow \varnothing$
    $\mathsf{B}_{\Pi_0^L} \leftarrow \textsc{Const}(0)$
    $i \leftarrow 0$
    foreach $\mathsf{B} \in B$ do
        $\mathsf{B}_{\Pi_0^L} \leftarrow \mathsf{B}_{\Pi_0^L} \vee (\mathsf{B} \wedge \textsc{EncodeBlock}(i))$
    return $\mathsf{B}_{\Pi_0^L}$
```

and exit rates as simple as

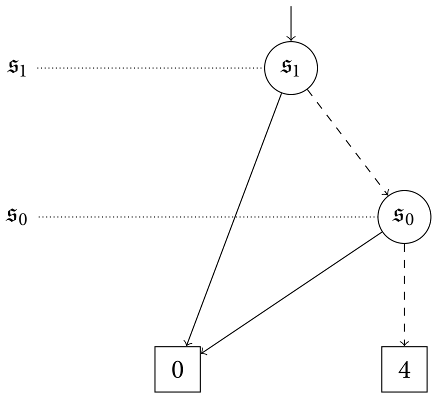
$$\Pi_0^{\mathcal{M}} = \textsc{Refine}(\mathsf{M}_r, \textsc{Refine}(\mathsf{M}_E, \mathsf{B}_{\Pi_0^L}, \mathcal{S}, |\Pi_0^L|), \mathcal{S}, b'),$$

where b' is the number of blocks of the intermediate partition that respects the state labels and exit rates but not the state rewards.

Example 45. For the MRA of Example 32, the partition Π_0^L maps all states to the single block with number 0. The BDD representation of this partition is given in Figure 5.9(a). Similarly, the exit rate MTBDD M_E is given in Figure 5.9(b) where we choose to make the 0-leaf explicit for illustration. The two leaves are the only nodes in M_E that are not labeled with a state variable. Refine therefore assigns states to different blocks that disagree on the exit rates, which, in particular, separates probabilistic from Markovian states. Consequently, Refine yields the partition $\Pi_0^{\mathcal{M}}$



(a) The partition BDD $\mathsf{B}_{\Pi_0^L}$ for the example MRA $\mathcal{M}$.



(b) The MTBDD M_E for the example MRA $\mathcal{M}$.

Figure 5.9: The BDDs used for creating $\Pi_0^{\mathcal{M}}$ for the example MRA $\mathcal{M}$.

with the BDD representantion as given in Figure 5.4.

5.4.4 Symbolic Implementation of the sigref Operator for MRA

We start by illustrating that Algorithm 3 does not yield the desired result for nondeterministic models such as PA and MA when encoded as in Section 2.5. In contrast to the representation of DTMCs, the transition relation MTBDD M_Δ also needs to encode the *nondeterminism* of the model. In general, it uses the variables $Var_{\mathcal{M}} = \mathcal{S} \uplus \mathcal{S}' \uplus \mathcal{A} \uplus \mathcal{N}$ (see beginning of Section 5.4). More specifically, is uses the variables $\mathcal{N}$ to disambiguate the individual distributions. In contrast to the MC case, there may be several nodes in $\mathsf{M}^{\Pi}_{\text{sig}}$ representing the same signature when the signature is computed as in (5.1) on page 128.

Example 46. Computing the signature MTBDD for $\mathcal{M}$ of Example 32 with respect to the partition $\Pi_0^{\mathcal{M}}$ whose BDD is given in Figure 5.4 yields the MTBDD in Figure 5.10. Even though the states s_2 and s_3 have the same signature (see Example 38), their encodings lead to different nodes p_2 and p_3. Applying REFINE to this MTBDD as before therefore splits the two states and ends up with the trivial state partition that assigns each state to its own block.

The problem is caused by the fact that the nondeterminism variables $\mathcal{N}$ are used to distinguish the different distributions available in a state. They are an artifact of the fact that DDs represent functions whereas the choices of a nondeterministic model are in fact a relation. Applying Algorithm 3 to such a model does separate states that have the same signature but use different encodings over $\mathcal{N}$ to encode their distributions. The actual result depends on the concrete nondeterminism encoding, whereas $\sim$ is clearly independent of this.

One way to approach this problem is to extend Algorithm 2 to deal with the nondeterminism encodings. In [Deh11], the authors propose an extension that relies on a nested depth-first search within the regular REFINE procedure. Computing the bisimulation relation this way, however, comes at the expense of computational overhead. First, the nested search may visit nodes multiple times, which impacts running times negatively. And secondly, the used signature cache needs to be queried for *sets of nodes* rather than just individual nodes. Intuitively, this set of nodes arises from abstracting from the nondeterminism encodings for a particular state. In practice, the basic operations used on the cache (see Algorithm 2) become more expensive and make partition refinement computationally costly [Deh11]. Furthermore, this technique requires a more restricted variable ordering that may increase the sizes of the DDs.

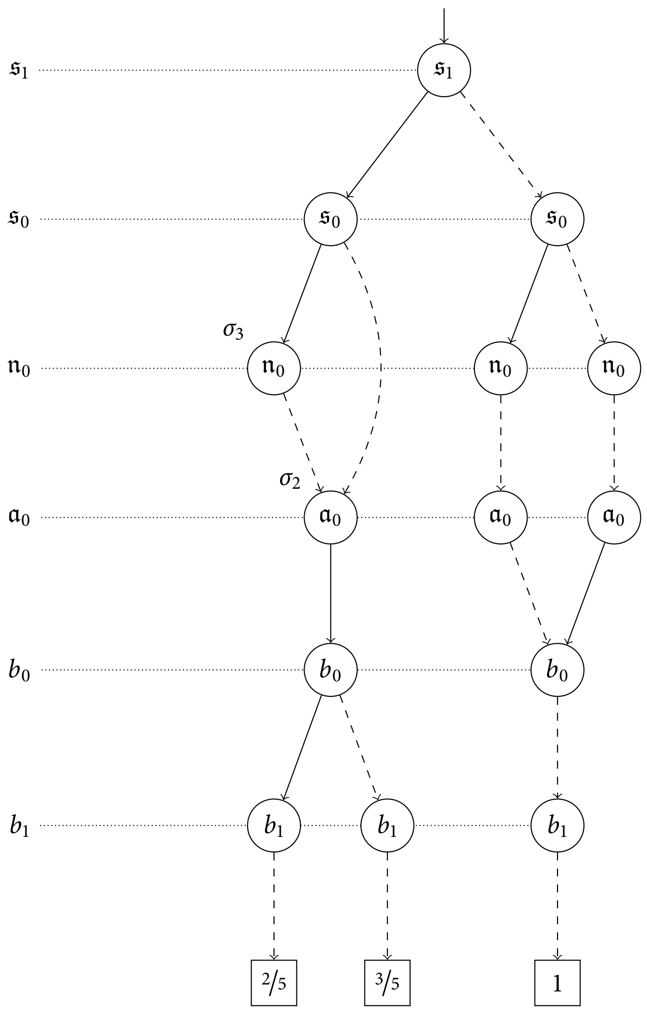


Figure 5.10: The signature MTBDD for the example $\mathcal{M}$ with respect to $\Pi_0^{\mathcal{M}}$.

Learning from the problems with this approach, we propose a different two-step algorithm. Its core idea is very simple: based on the observation that the intricacies stem from mixing qualitative choice (nondeterminism) with probabilistic choice (the distributions), we keep the two apart. Instead of maintaining just a partition Π of the states, we *additionally maintain a partition* Π^d *of the (rewarded) distributions available as nondeterministic choices in the states.* This is based on the following decomposition of the state signature.

Definition 38 (Signature Decomposition)**.** Let $S^d = \{\langle \rho, \mu \rangle \mid \exists s, \alpha \,.\, \langle s, \alpha, \rho, \mu \rangle \in \Delta\}$ be the (rewarded) distributions of $\mathcal{M}$. The signature of a rewarded distribution $\langle \rho, \mu \rangle$ with respect to a partition Π of S is given by

$$\mathrm{dsig}_\Pi(\rho, \mu) = \langle [\rho, \mu]_\Pi, [\mu]_\Pi \rangle \,.$$

By construction, dsig_Π induces a distribution partitioning Π^d of S^d by

$$\Pi^d = \Big\{ \big\{ \langle \rho', \mu' \rangle \in S^d \mid \mathrm{dsig}_\Pi(\rho', \mu') = \mathrm{dsig}_\Pi(\rho, \mu) \big\} \mid \langle \rho, \mu \rangle \in S^d \Big\}$$

The *choice signature* of a state $s \in S$ with respect to Π is defined by

$$\mathrm{csig}_\Pi(s) = \big\{ \langle \alpha, B \rangle \mid B \in \Pi^d \wedge \exists \langle \alpha, \rho, \mu \rangle \in \Delta(s) \,.\, \langle \rho, \mu \rangle \in B \big\} \,.$$

Intuitively, the choice signature of a state s represents which equivalence classes of Π^d are covered with which action label $\alpha \in \mathit{Act}$ in s. Whenever there are no transition rewards, we omit the component ρ for better readability.

Example 47. Reconsider the MRA $\mathcal{M}$ of Example 32 and the partition Π induced by the equivalence relation $\mathcal{R}$ of Example 33. Slightly abusing the notation to account for the missing transition rewards, $\mathrm{dsig}_\Pi(\mu_{2,1}) = \mu_{2,1}^{\mathcal{R}} = [\mu_{2,1}]_{\mathcal{R}}$. The distribution partition induced by dsig_Π is

$$\Pi^d = \left\{ \underbrace{\{\mu_0, \mu_1\}}_{B_0}, \underbrace{\{\mu_{2,1}, \mu_{2,2}, \mu_3\}}_{B_1} \right\} .$$

We therefore have

» $\mathrm{csig}_{\Pi}(s_0) = \{\langle \alpha, B_0 \rangle\}$,

» $\mathrm{csig}_{\Pi}(s_1) = \{\langle \Lambda, B_0 \rangle\}$, and

» $\mathrm{csig}_{\Pi}(s_2) = \mathrm{csig}_{\Pi}(s_3) = \{\langle \alpha, B_1 \rangle\}$.

Then, we have the following connection between the signature $\mathrm{sig}_{\Pi}(s)$ of a state s and its choice signature $\mathrm{csig}_{\Pi}(s)$.

Lemma 3. *Let $s, t \in S$ and Π be a partition of S.*

$$\mathrm{sig}_{\Pi}(s) = \mathrm{sig}_{\Pi}(t) \iff \mathrm{csig}_{\Pi}(s) = \mathrm{csig}_{\Pi}(t).$$

Considering csig_{Π} instead of sig_{Π} has the advantage that it decouples quantities (probabilities, rewards) from nondeterministic choices. While the signatures of rewarded distributions are quantitative in nature, they do not contain nondeterminism. In contrast, the choice signature of a state is purely qualitative. Note that a very similar decomposition idea was formulated in the context of an explicit state representation in [BEM00].

We use this connection and develop Algorithm 5 to symbolically implement the sigref operator for MRA. In correspondence to the choice signature formulation, it maintains two partitions: a state partition Π and a distribution partition Π^d that corresponds to the partition induced by dsig_{Π}. Note that both partitions share the same block variables $\mathcal{B}$. This is possible as enough block variables have been reserved to also represent the distribution partition (see Section 5.4.1). In lines 3 and 4, the counters b^S and b^d for the number of state and distribution blocks, respectively, are initialized. Line 5 initializes the distribution partition by assigning all distributions that appear in $\mathcal{M}$ to a single block. The first refinement step refines the distribution partition with respect to the quotient distributions (lines 8-9). For this, it computes a signature MTBDD representing $\mathrm{dsig}_{\Pi_{i-1}}$ and calls Refine similarly as before. However, there is one important difference, namely that instead of just the variables $\mathcal{S}$ now also the variables $\mathcal{A}$ and $\mathcal{N}$ need to be considered as "state variables" by Refine. This reflects that the partition that is created is a partition of (rewarded) distributions rather than states. As MRA may also involve transition rewards, the next step refines the distribution partition, however this time with respect to the quotient reward functions. Line 10 computes a representation of the quotient reward functions as in Definition 32 and line 11 performs the corresponding refinement step of the just computed partition BDD $\mathsf{B}_{\Pi^d_i}$. Note that the computation of the quotient reward functions reuses the distribution signature MTBDD. In summary, the lines 8 through 11 are concerned with refining the distribution partition according to $\mathrm{dsig}_{\Pi_{i-1}}$.

Algorithm 5: Symbolic signature-based partition refinement for MRA.

```
1  function SignatureRefinement(MRA $\mathcal{M}$, partition $\Pi_0$):
       input: $\mathcal{M}$: the MRA for which to compute $\sim_\mathcal{M}$,
              $\Pi_0$: the initial partition
       output: the bisimilarity relation $\sim_\mathcal{M}$ represented by a BDD $\mathsf{B}_{\Pi_i}$
2      $i \leftarrow 0$
3      $b^S \leftarrow |\Pi_0|$
4      $b^d \leftarrow 0$
       // create trivial distribution partition (and increment $b^d$)
5      $\Pi^d_0 = \textsc{ExistsAbstract}\,(\mathcal{S}', \mathsf{M}_\Delta \neq 0) \wedge \textsc{EncodeBlock}(b^d)$
6      repeat
7          $i \leftarrow i + 1$
           // refine $\Pi^d$ w.r.t. quotient distributions
8          $\mathsf{M}_{\text{sig}} \leftarrow \textsc{SumAbstract}\,(\mathcal{S}', \mathsf{M}_\Delta \cdot \textsc{Rename}\,(\mathsf{B}_{\Pi_{i-1}}, \mathcal{S}, \mathcal{S}'))$
9          $\mathsf{B}_{\Pi^d_i} \leftarrow \textsc{Refine}(\mathsf{M}_{\text{sig}}, \mathsf{B}_{\Pi^d_{i-1}}, \mathcal{S} \uplus \mathcal{A} \uplus \mathcal{N}, b^d)$
           // refine $\Pi^d$ w.r.t. quotient reward functions
10         $\mathsf{M}_{\text{sig}} \leftarrow \textsc{SumAbstract}\left(\mathcal{S}', \mathsf{M}_\Delta \cdot \mathsf{M}^\rho_\Delta \cdot \textsc{Rename}\,(\mathsf{B}_{\Pi_{i-1}}, \mathcal{S}, \mathcal{S}')\right) / \mathsf{M}_{\text{sig}}$
11         $\mathsf{B}_{\Pi^d_i} \leftarrow \textsc{Refine}(\mathsf{M}_{\text{sig}}, \mathsf{B}_{\Pi^d_i}, \mathcal{S} \uplus \mathcal{A} \uplus \mathcal{N}, b^d)$
           // refine $\Pi$ w.r.t. $\Pi^d$
12         $\mathsf{B}_{\text{sig}} \leftarrow \textsc{ExistsAbstract}\left(\mathcal{N}, \mathsf{B}_{\Pi^d_i}\right)$
13         $\mathsf{B}_{\Pi_i} \leftarrow \textsc{Refine}(\mathsf{B}_{\text{sig}}, \mathsf{B}_{\Pi_{i-1}}, \mathcal{S}, b^S)$
14     until $\mathsf{B}_{\Pi_i} = \mathsf{B}_{\Pi_{i-1}}$
15     return $\mathsf{B}_{\Pi_i}$
```

Finally, lines 12 and 13 refine the state partition with respect to Π^d_i. This corresponds directly to the computation of a signature BDD for csig_{Π_i} and an application of Refine.

Example 48. The MTBDD of Figure 5.10 represents the signatures of the distributions of $\mathcal{M}$, where the latter are encoded over $\mathcal{S} \uplus \mathcal{N} \uplus \mathcal{A}$. Using this MTBDD and the trivial distribution partition Π^d_0 that assigns all distributions to block 0 as the previous partition, Refine yields the distribution partition Π^d of Example 47 in the form of the BDD depicted in Figure 5.11. To compute the BDD for the choice signatures of the states, we now existentially abstract from the variables in $\mathcal{N}$. This results

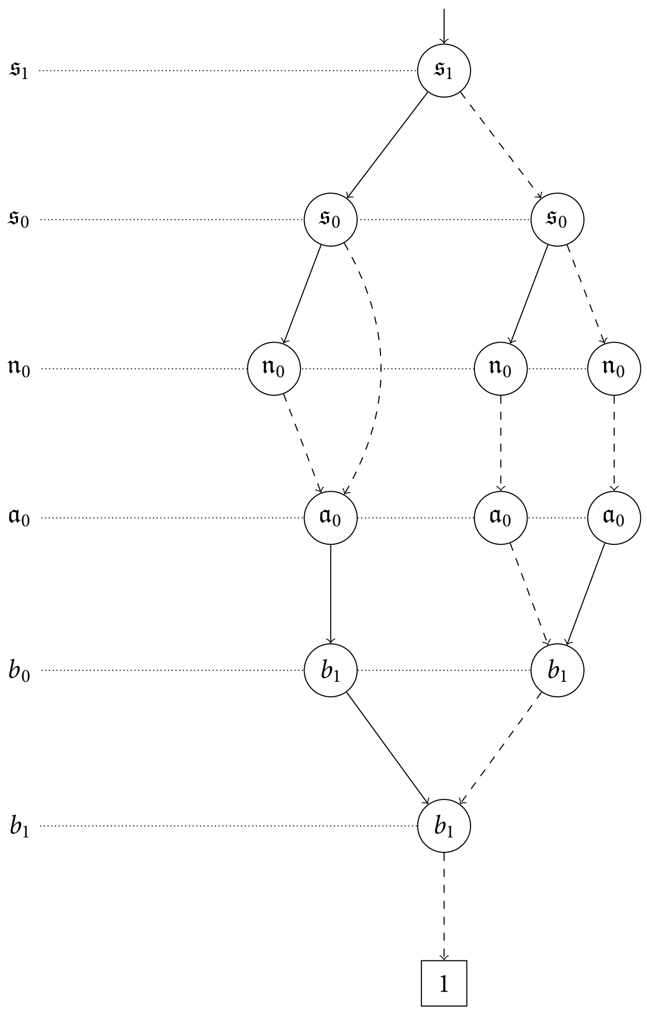


Figure 5.11: The BDD representing the distribution partition Π^d.

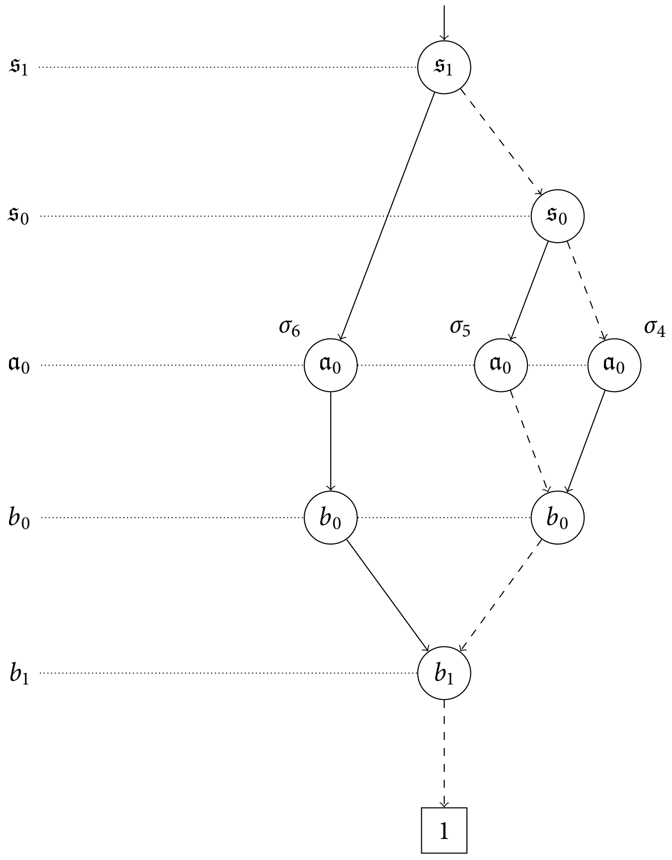


Figure 5.12: The BDD representing the choice signatures of the states.

in the BDD in Figure 5.12. Invoking Refine on this BDD together with the previous state partition BDD of Example 37 yields the (final) partition $S/{\sim}$ represented by the BDD in Figure 5.8.

5.5 Quotienting

As our ultimate goal is to verify or refute a property on the bisimulation quotient rather than on the original MRA, it remains to show how the quotient can be extracted from the symbolic representation of $\mathcal{M}$ and a BDD $\mathsf{B}_{\sim}$ that represents the (state) partitioning induced by $\sim$ in our partition representation (see Section 5.4.1).

5.5.1 Symbolic quotienting

Recall the symbolic representation of the MRA $\mathcal{M}$ from the start of the section and Definition 35 for the quotient of $\mathcal{M}$ under a bisimulation $\mathcal{R}$. We now derive a symbolic representation of $\mathcal{M}/{\sim}$. In contrast to the variant (for transition systems) shown in [Wim10], we use the variables $\mathcal{B}$ for the encoding of states rather than using the original variables $\mathcal{S}$. This might change the relative order of the variables encoding the nondeterminism to the (new) variables encoding the states. While this may affect the sizes of the DDs, our representation can be trivially shifted to the original variables of the MRA representation like [Wim10]. However, renaming variables tends to be an expensive operation and our experiments show that using the block variables for state representation does not have adverse effects. As we use the block variables for the state encoding, we assume an additional set of variables $\mathcal{B}' = \{b' \mid b \in \mathcal{B}\}$ that encode the successor states.

As a first step, we present an algorithm that takes a BDD representing a state partition and returns a BDD representing a set of representatives such that for each block there is exactly one representative. Algorithm 6 shows the pseudo-code of this routine. It maintains an initially empty set *visited* of DD nodes that have been seen in the traversal. Invoked with SelectRepresentatives($\mathsf{B}_{\Pi}, \mathcal{S}$), it recursively descends depth-first through the partition BDD B_{Π} until a node is found that is not labeled with a state variable $\mathcal{S}$. In this case, the node must correspond to exactly one block encoding as B_{Π} adheres to our partition representation scheme (see Section 5.4.1). To not select multiple representatives for some block, attention is paid to insert a unique valuation of skipped variables (lines 6 to 9) and not visiting the same node more than once (line 2). Technically, it is sufficient to only remember the block encodings that were seen in the *visited* set, but aborting the depth-first search upon a revisit of any node improves the efficiency as it potentially avoids visiting large parts of B_{Π}.

Algorithm 6: Selection of representative states.

```
1  function SelectRepresentatives($\sigma$, $Var$):
       input: $\sigma$: the root node of the partition BDD $\mathsf{B}_\sim$,
              $Var$: the state variables
       output: a BDD containing exactly one state of each block
2      if $\sigma \in visited$ then
3          return Const $(0)$
4      $x \leftarrow \min\{Var \cup \{var(\sigma)\}\}$
5      if $x \in Var$ then
6          if $x = var(\sigma)$ then
7              $Var' \leftarrow Var \setminus \{x\}$
8          else
9              $Var' \leftarrow Var$
10         $low \leftarrow$ SelectRepresentatives$(\sigma|_{x=0}, Var')$
11         $high \leftarrow$ SelectRepresentatives$(\sigma|_{x=1}, Var')$
12         $result \leftarrow$ Ite $(x, high, low)$
13     else
14         $result \leftarrow$ Const $(1)$
15     $visited \leftarrow visited \cup \{\sigma\}$
16     return $result$
```

In the following, we assume that the BDD B_{Rep} of representative states over the variables $\mathcal{S}$ has been computed as $\textsc{SelectRepresentatives}(\mathsf{B}_\Pi, \varnothing, \mathcal{S})$ where $\mathsf{B}_\sim$ is the partition BDD obtained by $\textsc{SignatureRefinement}(\mathcal{M}, \Pi_0^{\mathcal{M}})$.

Let us now recall Definition 35. Several components of the quotient MRA can be computed in a straightforward manner:

» $\mathsf{B}_{S/\sim} = \textsc{ExistsAbstract}\,(\mathsf{B}_\sim, \mathcal{S})$,

» $\mathsf{B}_{S^0/\sim} = \textsc{ExistsAbstract}\,\left(\mathsf{B}_{S^0} \wedge \mathsf{B}_\sim, \mathcal{S}\right)$,

» $\mathsf{B}_{a/\sim} = \textsc{ExistsAbstract}\,(\mathsf{B}_a \wedge \mathsf{B}_\sim, \mathcal{S})$ for all atomic propositions $a \in AP$,

» $\mathsf{M}_{E/\sim} = \textsc{SumAbstract}\,\left(\mathsf{M}_E \cdot \mathsf{B}_{\text{Rep}} \cdot \mathsf{B}_\sim, \mathcal{S}\right)$, and

» $\mathsf{M}_{r/\sim} = \textsc{SumAbstract}\,\left(\mathsf{M}_r \cdot \mathsf{B}_{\text{Rep}} \cdot \mathsf{B}_\sim, \mathcal{S}\right)$.

It remains to compute the MTBDDs $\mathsf{M}_{\Delta/\sim}$ and $\mathsf{M}^{\rho}_{\Delta/\sim}$.

The quotient probability distributions can be computed in two steps.

$$\begin{aligned}
\mathsf{M}^{\mathcal{B}'}_{\Delta} &= \textsc{SumAbstract}\left(\mathsf{M}_{\Delta} \cdot \textsc{Rename}\left(\mathsf{B}_{\sim}, \mathcal{S} \uplus \mathcal{B}, \mathcal{S}' \uplus \mathcal{B}'\right), \mathcal{S}'\right) \\
\mathsf{M}_{\Delta/\sim} &= \textsc{SumAbstract}\left(\mathsf{M}^{\mathcal{B}'}_{\Delta} \cdot \mathsf{B}_{\mathrm{Rep}} \cdot \mathsf{B}_{\sim}, \mathcal{S}\right)
\end{aligned}$$

Here, $\mathsf{M}^{\mathcal{B}'}_{\Delta}$ is over the variables $\mathcal{S} \uplus \mathcal{A} \uplus \mathcal{N} \uplus \mathcal{B}'$ and is the intermediate result that specifies for each nondeterministic choice in each state s the probability to move to the blocks of $\sim$. In the second step, a representative state of each block is used to determine the behavior of the block. Similarly, we compute $\mathsf{M}^{\rho}_{\Delta/\sim}$ as

$$\begin{aligned}
\mathsf{M}^{\rho,\mathcal{B}'}_{\Delta/\sim} &= \textsc{SumAbstract}\left(\mathsf{M}^{\rho}_{\Delta} \cdot \mathsf{M}_{\Delta} \cdot \textsc{Rename}\left(\mathsf{B}_{\sim}, \mathcal{S} \uplus \mathcal{B}, \mathcal{S}' \uplus \mathcal{B}'\right), \mathcal{S}'\right) / \mathsf{M}^{\mathcal{B}'}_{\Delta} \\
\mathsf{M}^{\rho}_{\Delta/\sim} &= \textsc{SumAbstract}\left(\mathsf{M}^{\rho,\mathcal{B}'}_{\Delta/\sim} \cdot \mathsf{B}_{\mathrm{Rep}} \cdot \mathsf{B}_{\sim}, \mathcal{S}\right).
\end{aligned}$$

The reward functions need to be scaled with the total probability to move to an equivalence class (see Definition 32) and we can reuse the intermediate BDD $\mathsf{M}^{\mathcal{B}'}_{\Delta}$ for the corresponding division.

5.5.2 Direct Sparse Extraction

It is well known [BS92] that bisimulation quotienting might *increase* the number of nodes that is necessary to symbolically represent the system, an effect that we will further encounter in our experimental evaluation in Section 5.6. Briefly speaking, symbolic data structures are well-suited to represent symmetrical systems, but bisimulation minimization factors out the symmetry and leaves a system that is largely asymmetrical. This effect suggests that the state space savings due to this minimization do not translate into speedups or improved memory footprints. In fact, the opposite is true and it appears to render bisimulation minimization void. However, there is an avenue worth pursuing similar to that proposed in [Par03]. Instead of performing all operations symbolically, we only treat it symbolically up to the point where we want to conduct model checking. Then, the symbolic transition relation is translated to an explicit representation on which the numerical analysis is carried out. Since for an explicit representation reductions in state space size are more promising to directly translate into performance gains, this approach may not suffer too much from the increased DD sizes.

However, as explicitly mentioned in [Wim10], extracting the quotient is a computationally costly operation. Intuitively, this is because the intermediate DDs (partitions, signatures, etc.) relate variables that are far apart in the variable ordering, which causes them to be large on the one hand and requires an expensive renaming to return to an interleaved (with respect to source/target states) variable ordering. Ultimately, this problem boils down to the chosen representation of a partition over state variables that precede all block variables. However, the central refinement procedure requires this variable order in order to exploit the reducedness property of the DDs.

Instead of first computing the symbolic representation of the quotient and *then* converting it to an explicit representation, we propose *to extract the explicit quotient directly*. Since the extraction of the quotient transition relation is the most involved part, we restrict our attention to this setting. The other components of the system can be derived similarly.

Algorithm 7 shows the pseudo code of the procedure Extract. It returns a function

$$f\colon \{0,\dots,\mathrm{ind}(\sim)\} \times \mathit{Act} \times \mathbb{N} \times \{0,\dots,\mathrm{ind}(\sim)\} \to [0,1]$$

that represents $\Delta/\sim$. It does so in the sense that

$$\langle s,\alpha,\mu\rangle \in \Delta/\sim \iff \exists n \in \mathbb{N}\,.\, f(s,\alpha,n,\cdot) = \mu$$

where we identify states of the quotient system with the numbers of their block within the partition $\sim$. The additional parameter n can be thought of as an index that uniquely identifies the distribution among all those labeled with the same action α. From this representation it is straightforward to extract other representations, for example in the form of sparse matrices.

Let us now step through the building blocks of the algorithm. The procedure takes 7 arguments:

» m is the current node in the transition relation MTBDD,

» r is the current node in the representatives BDD $\mathsf{B}_{\mathrm{Rep}}$,

» p_s is the node in the partition BDD $\mathsf{B}_\sim$ corresponding to the current source state,

» p_t is the node in the partition BDD $\mathsf{B}_\sim$ corresponding to the current target state,

» Var_s is the set of (remaining) state variables,

» B_n is a BDD that encodes the current nondeterminism variables' valuation, and

» Var_n is the set of (remaining) nondeterminism variables.

Algorithm 7: Direct sparse extraction of the quotient transition relation $\Delta/\sim$.

```
function EXTRACT($m, r, p_s, p_t, Var_s, \mathsf{B}_n, Var_n$):
    input: $m$: the root node of the transition relation MTBDD $\mathsf{M}_\Delta$,
           $r$: the root node of the BDD containing the representative states,
           $p_s$: the root node of the partition BDD,
           $p_t$: the root node of the partition BDD,
           $Var_s$: the state variables,
           $\mathsf{B}_n$: the nondeterminism encoding (initially CONST (1)),
           $Var_n$: the nondeterminism variables
    output: a function explicitly encoding the choices of the quotient model
    if $r$ = CONST (0) then                        // no representative state, abort
        return $\varnothing$
    if $Var_s = \varnothing \wedge Var_n = \varnothing$ then    // $m$ is terminal node of transition relation
        $s \leftarrow$ DECODEBLOCK($p_s$)
        $t \leftarrow$ DECODEBLOCK($p_t$)
        $\alpha \leftarrow$ DECODEACTION($\mathsf{B}_n$)
        $n \leftarrow$ DECODEOFFSET($\mathsf{B}_n$)
        return $\langle s, \alpha, n, t\rangle \mapsto val(m)$            // add explicit transition

    $x \leftarrow \min\{Var_s \cup Var_n\}$
    if $x \in Var_s$ then
        // descend by assigning both state variables $x$ and $x'$
        $f_{00} \leftarrow$ EXTRACT($m|_{x=0,x'=0}, r|_{x=0}, p_s|_{x=0}, p_t|_{x=0}, Var_s \setminus \{x\}, \mathsf{B}_n, Var_n$)
        $f_{01} \leftarrow$ EXTRACT($m|_{x=0,x'=1}, r|_{x=0}, p_s|_{x=0}, p_t|_{x=1}, Var_s \setminus \{x\}, \mathsf{B}_n, Var_n$)
        $f_{10} \leftarrow$ EXTRACT($m|_{x=1,x'=0}, r|_{x=1}, p_s|_{x=1}, p_t|_{x=0}, Var_s \setminus \{x\}, \mathsf{B}_n, Var_n$)
        $f_{11} \leftarrow$ EXTRACT($m|_{x=1,x'=1}, r|_{x=1}, p_s|_{x=1}, p_t|_{x=1}, Var_s \setminus \{x\}, \mathsf{B}_n, Var_n$)
        return $f_{00} + f_{01} + f_{10} + f_{11}$
    else
        // descend by assigning just the nondeterminism variable $x$
        $f_0 \leftarrow$ EXTRACT($m|_{x=0}, r, p_s, p_t, Var_s, \mathsf{B}_n \wedge$ ID ($x$), $Var_n \setminus \{x\}$)
        $f_1 \leftarrow$ EXTRACT($m|_{x=1}, r, p_s, p_t, Var_s, \mathsf{B}_n \wedge \neg$ID ($x$), $Var_n \setminus \{x\}$)
        return $f_0 + f_1$
```

Therefore, the top-level invocation is $\textsc{Extract}(\mathsf{M}_\Delta, \mathsf{B}_{\mathrm{Rep}}, \mathsf{B}_\sim, \mathsf{B}_\sim, \mathcal{S}, \textsc{Const}(1), \mathcal{A} \uplus \mathcal{N})$. The idea is to recursively walk through the transition relation MTBDD M_Δ and keep track of which source and target blocks the transitions connect by *simultaneously* traversing the partition BDD $\mathsf{B}_\sim$. As both the source and the target blocks need to be determined, the algorithm maintains two nodes of the partition BDD, namely p_s that corresponds to the block of the source state and p_t that corresponds to the block of the target state. Since we only want to extract the transitions of the representative state of each block, no quotient transition is recorded if the currently followed state encoding does not belong to a representative state (line 2). If a terminal node has been found in the transition relation M_Δ (line 4), the nodes p_s and p_t allow for easy discovery of the source and target blocks. Additionally, the BDD B_n over the variables $\mathcal{A} \uplus \mathcal{N}$ uniquely identifies the action label and index of the distribution and can therefore be used to identify the correct choice. We can therefore record the quotient transition by creating the appropriate mapping in line 9. Lines 11 to 21 take care of the recursive descent. If the next variable is a variable encoding a state, we assign to the variable and its primed counterpart and move in both p_s and p_t. If, however, the next variable is a variable encoding nondeterminism, this is reflected by updating the transition relation node and the BDD B_n encoding the current nondeterminism encoding. In both recursive cases, the remaining variable sets are updated.

While our algorithm is independent of the variable ordering, an efficient implementation requires that the variable sets $\mathcal{S}$ and $\mathcal{S}'$ are ordered in an interleaving manner, because the generalized cofactors then amount to simply following pointers. This is not a real restriction, though, as most tools choose to arrange the variables like this.

We want to remark that actual implementations might look slightly differently than the pseudo code in Algorithm 7 for performance reasons. For example, it is possible to use offset-labeled binary decision diagrams (ODDs) [Par03] to decode source and target states during the traversal.

5.6 Evaluation

Implementation. We implemented symbolic bisimulation minimization within the framework of Storm (see Chapter 7). It is available using both major libraries that support MTBDDs, namely CUDD [Som] and Sylvan [Dij16]. The Sylvan-based implementation reuses substantial parts from the implementation of [DP18] and supports multi-threaded signature computation and refinement. Additionally, it can not only deal with probabilities stored as floating point numbers but also as rational numbers and even rational functions.

Unlike our presentation, our implementation does not use $\Pi_0^{\mathcal{M}}$ as the initial partition. Instead, we assume a set of properties to be available that are to be checked. We then extract the atomic propositions used in these formulae and additionally determine whether they refer to rewards or not. The initial partition can then be created with respect to the labels appearing in the properties and does not need to be refined according to the rewards if the property does not refer to them. This can significantly influence the size of the resulting bisimulation quotient. In the context of the evaluation, we therefore always consider pairs of model instances and properties.

Benchmarks. For the evaluation of the prototype we considered 12 benchmark models covering DTMCs, CTMCs, PA and MRA. With the exception of the MRA, they are part of PRISM's benchmark suite [KNP12]. The MRA models are taken from [QJK17] and [Guc+13]. Both, the models and the properties are listed in more detail in Appendix D. Similarly, Appendix D contains more detailed experimental results. As previous work [Wim10] determined that floating point arithmetic may be harmful to symbolic bisimulation minimization in the sense that it can lead to unnecessarily large quotients and even non-terminating behaviour of the refinement algorithm, we choose to use rational arithmetic and therefore focus on our SYLVAN-based implementation. To not introduce unwanted side-effects due to multi-threading, we limit all runs to a single thread.

State space reduction. We start by showing the state space reductions that can be obtained on the models. Table 5.1 lists for each benchmark instance the size of the model in terms of states prior to minimization and the size of the quotient system. With few exceptions (`polling`) the considered models are reduced significantly by bisimulation minimization. Disregarding the models that could not be reduced at all, the reduction factors over all models range from roughly 4 for the `jobs(15,3)` to more than 10^{17} for `crowds(30,30)`. For the latter, neither SYLVAN nor CUDD could count the number of minterms of the state and transition relation DDs for the original model. We therefore give lower bounds for the state and transition count based on a smaller instance for which counting states and transitions does not (yet) fail. In contrast, the reduced state model has just below 600 states. A similarly huge reduction can be observed for the CTMC benchmark `p2p`: its quotient model is roughly 9 orders of magnitude smaller than the original model.

In general, however, we observe that continuous-time models tend to be less symmetrical in the sense of strong bisimulation. The reason for this is that for two states to be considered equivalently not only their probabilistic behavior has to match, but also their timing behavior. This separates states that could have been merged in discrete-time

			original		quotient	
	model	instance	states	transitions	states	transitions
DTMC	bluetooth	(1)	3.4×10^{9}	5.0×10^{9}	3.7×10^{2}	3.7×10^{2}
		(2)	5.5×10^{10}	5.9×10^{10}	7.8×10^{5}	1.4×10^{6}
	crowds	(30, 20)	6.1×10^{15}	3.0×10^{16}	3.8×10^{2}	5.7×10^{2}
		(30, 30)	$> 1.3 \times 10^{19}$	$> 1.4 \times 10^{19}$	5.8×10^{2}	8.7×10^{2}
	leader	(6, 8)	1.3×10^{6}	1.6×10^{6}	1.4×10^{1}	1.5×10^{1}
		(7, 7)	4.9×10^{6}	5.8×10^{6}	1.6×10^{1}	1.7×10^{1}
CTMC	embedded	(1000)	8.5×10^{5}	3.6×10^{6}	6.5×10^{4}	4.0×10^{5}
		(2000)	1.7×10^{6}	7.1×10^{6}	1.3×10^{5}	7.9×10^{5}
	polling	(16)	1.6×10^{6}	1.4×10^{7}	1.6×10^{6}	1.4×10^{7}
		(17)	3.3×10^{6}	3.1×10^{7}	3.3×10^{6}	3.1×10^{7}
	p2p	(7, 5)	3.4×10^{10}	6.0×10^{11}	1.1×10^{3}	3.9×10^{3}
		(8, 5)	1.1×10^{12}	2.2×10^{13}	1.4×10^{3}	4.8×10^{3}
PA	coin	(6, 4)	2.4×10^{6}	1.2×10^{7}	5.2×10^{3}	3.1×10^{4}
		(6, 6)	3.5×10^{6}	1.7×10^{7}	7.7×10^{3}	4.6×10^{4}
	csma	(3, 4)	1.5×10^{6}	2.4×10^{6}	3.0×10^{4}	6.4×10^{4}
		(4, 4)	1.3×10^{8}	2.6×10^{8}	3.9×10^{5}	1.0×10^{6}
	wlan_dl	(7, 140)	8.1×10^{8}	1.9×10^{9}	4.6×10^{5}	1.3×10^{6}
		(8, 140)	2.2×10^{9}	5.1×10^{9}	4.6×10^{5}	1.3×10^{6}
MA	mutex	(10)	1.1×10^{6}	3.0×10^{6}	1.3×10^{5}	3.7×10^{5}
		(15)	3.6×10^{6}	9.7×10^{6}	4.0×10^{5}	1.2×10^{6}
	polling	(3, 4)	9.1×10^{4}	2.3×10^{5}	9.1×10^{4}	2.3×10^{5}
		(4, 4)	8.3×10^{5}	2.1×10^{6}	8.3×10^{5}	2.1×10^{6}
	jobs	(15, 3)	1.9×10^{6}	7.5×10^{6}	4.6×10^{5}	2.0×10^{6}
		(16, 3)	4.7×10^{6}	1.8×10^{7}	9.0×10^{5}	3.9×10^{6}

Table 5.1: The sizes of the bisimulation quotients for the considered benchmark models.

models. As an extreme case, for the two `polling` models (one is a CTMC and the other one is an MA), no reduction is possible at all: the quotient models are just as large as the original model.

Symbolic verfication. While the state space reductions can be huge, we now analyze whether they translate to gains in the model checking process. In the following, we therefore compare two approaches: verifying the original model directly versus first constructing the bisimulation quotient and then verifying this instead. Since the model-building step is the same for both, we omit these figures from the evaluation. Note that construction of the quotient consists of two phases, namely (symbolic) partition refinement to compute the bisimulation and extracting the quotient from the original model and $\sim$. For each experiment, we set a timeout of 1 hour and a memory limit of 16GB and set the maximal memory to be used by Sylvan to 8GB.

As a first step, we measure the impact of bisimulation minimization on DD-based verification. Storm's support for this is limited to discrete-time Markov models at the moment, since the analysis of continuous-time models tends to involve more costly numerical operations for which DDs are not well suited. Consequently, we consider all DTMCs and PA from our example set for this evaluation. Here, the bisimulation minimization as well as the actual verification are carried out on DD representations of the systems. The results are summarized in Table 5.2.

With the notable exception of both `crowds` and the `leader(7,7)` instances, the state space reductions do not carry over to the overall runtimes. To the contrary, the total time needed to verify the `wlan_dl` models increases fortyfold even though the state space is reduced by four orders of magnitude. Even the raw verification times (i. e. without considering the time taken by the minimization) are increased by a factor of thirty. Ultimately, this is caused by the blow-up of the DD representation of the quotient model: the quotient transition relation has roughly ten times the number of nodes that were necessary to store the original transition relation. To see the effect of bisimulation quotienting on the size of systems in terms of DD nodes, consider Figures 5.13 and 5.14.

In the majority of cases, the DD-based representation of the quotient used *more* nodes than the original transition relation. In particular, for the models involving nondeterminism (i. e. PA and MA), not a single instance had a smaller quotient representation. The blow-up ranges from a very minor increase (`jobs(16,3)`) to four orders of magnitude (`polling(17)`). This effect suggests that symbolic model checking rarely benefits from symbolic bisimulation minimization in its current form.

			original	quotient	
	model	instance	verification	construction	verification
DTMC	`bluetooth`	(1)	1.87	7.12	0.50
		(2)	TO	2130.19	TO
	`crowds`	(30, 20)	TO	152.11	96.71
		(30, 30)	TO	434.21	192.57
	`leader`	(6, 8)	96.85	123.28	0.02
		(7, 7)	536.54	496.50	0.02
PA	`coin`	(6, 4)	TO	35.77	TO
		(6, 6)	TO	55.34	TO
	`csma`	(3, 4)	1227.61	72.07	2125.60
		(4, 4)	TO	1922.98	TO
	`wlan_dl`	(7, 140)	55.09	748.65	1879.16
		(8, 140)	68.73	968.47	1722.64

Table 5.2: DD-based quotient model checking.

Hybrid verification. One particularly promising approach is the one that we touched in Section 5.5.2. First, the model is built and minimized symbolically. During the model checking step, however, the symbolic representation is translated into an explicit one and the latter is used for the expensive numerical operations. This approach [Par03] is pursued by the hybrid engine of STORM (for all model types except MA) and the sparse engine of PRISM. In the context of symbolic bisimulation minimization, it is appealing, because the state space reductions are more likely to translate into actual gains during the verification on explicit model representations. However, doing this translation without minimization as a preprocessing step potentially requires much more memory.

We therefore use STORM to model check the original models as well as the quotient models using its hybrid engine. As it is comparable in spirit, we additionally compare with the sparse quotient extraction of Section 5.5.2. The results of this comparison are displayed in Table 5.3.

We observe that sparse quotient extraction results in both reduced construction *and* verification times for *all* instances when compared to DD-based quotienting. For the

			original	quotient			
				DD quotient		sparse quotient	
	model	instance	verif.	constr.	verif.	constr.	verif.
DTMC	bluetooth	(1)	MO	7.07	0.05	6.16	<0.01
		(2)	MO	2513.92	530.63	2026.89	3.26
	crowds	(30, 20)	MO	152.07	0.13	134.87	<0.01
		(30, 30)	MO	432.73	0.21	392.84	0.01
	leader	(6, 8)	134.87	104.46	0.01	80.05	<0.01
		(7, 7)	MO	411.32	0.01	314.12	<0.01
CTMC	embedded	(1000)	1604.72	144.81	193.86	133.48	129.74
		(2000)	3004.38	746.98	445.19	489.59	283.05
	polling	(16)	1074.14	722.12	TO	280.95	1273.26
		(17)	2547.50	2955.19	TO	1581.84	TO
	p2p	(7, 5)	MO	283.93	0.42	224.72	0.02
		(8, 5)	MO	748.60	0.35	687.22	0.02
PA	coin	(6, 4)	1195.26	36.00	5.43	33.24	2.34
		(6, 6)	TO	55.93	12.40	52.34	6.21
	csma	(3, 4)	16.08	71.51	14.35	61.87	0.09
		(4, 4)	MO	2107.61	382.92	1598.20	2.04
	wlan_dl	(7, 140)	19.65	802.96	2068.94	652.36	1.09
		(8, 140)	20.07	814.42	1563.55	724.67	1.68

Table 5.3: Analysis of hybrid quotient model checking.

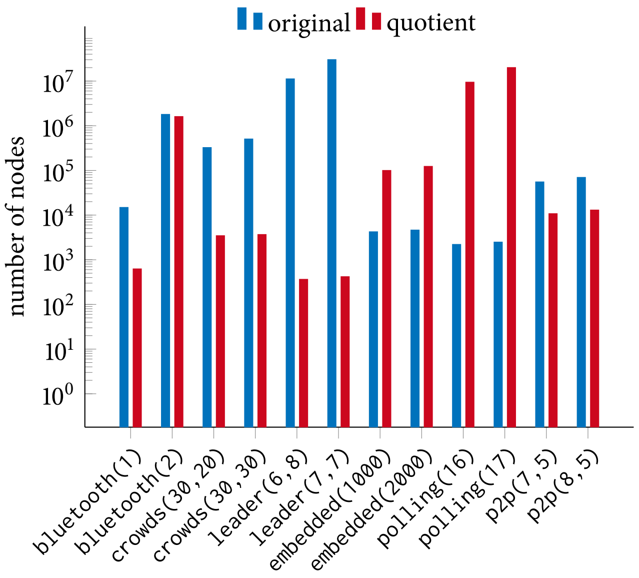


Figure 5.13: DTMC and CTMC benchmark models.

`polling(16)` CTMC benchmark, it reduced the time to extract the quotient from 488 seconds to less than 18 seconds and cut the overall time to construct the bisimulation quotient by two thirds. As it avoids the construction of the MTBDD $\mathsf{M}_{\Delta/\sim}$, it is not affected by the blow-up that symbolic quotienting incurs. Figures 5.15 and 5.16 summarize the runtime differences for the two quotient extraction approaches.

With the exception of `polling`, `wlan_dl` and the smaller `csma` instance, bisimulation minimizaton pays off. For a number of benchmark models, verification of the original model requires too much memory. Using bisimulation enables the treatment of all of these models with the hybrid approach. Even when there is no lack of memory, the bisimulation gains in terms of state space size translate into gains in terms of verification efficiency. Because of this, verifying most quotients is typically a matter of seconds.

As mentioned before, the `polling` CTMC benchmark does not allow for any minimization under strong bisimulation (using the property that we considered). Therefore, no speedups can be expected. Initially, we were surprised that the hybrid engine can treat the `wlan_dl` instances with more than 10^9 states whereas `csma(4,4)` led to a memory-

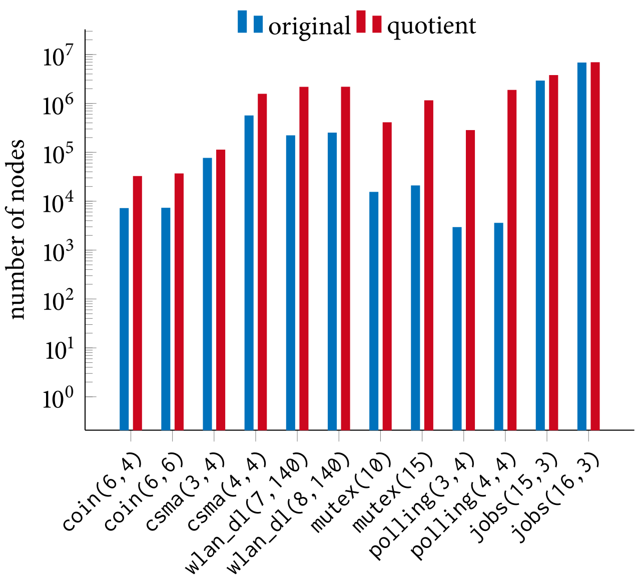


Figure 5.14: PA and MA benchmark models.

out even though it has fewer states and transitions. Not only that, but it is much faster than bisimulation minimization. Looking into the model, it turned out that almost all states related by bisimulation have a probability of zero for the considered property. This set of states is computed symbolically by the hybrid engine and, in particular, not translated to the explicit representation. Therefore, it doesn't require (structurally) more memory than verifying the quotient and the numerical computations are very limited even in the original model. We observe that the hybrid verification of the DD-based quotient takes almost three orders of magnitude longer than the verification of the sparse quotient. This effect is again related to the blow-up of the quotient transition relation MTBDD. While the numerical parts are solved quickly on the explicit representation, the qualitative precomputation suffers significantly from the increased DD sizes.

We turn to the impact of bisimulation minimization on the considered MA. As Storm currently does not support the symbolic or hybrid verification of such models, we resort to comparing the verification of the sparse model and verifying the quotient obtained through symbolic bisimulation minimization and direct sparse extraction in Table 5.4.

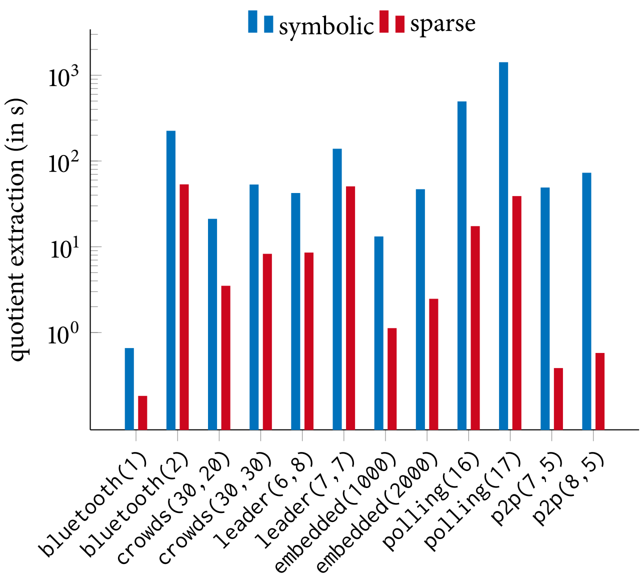


Figure 5.15: DTMC and CTMC benchmark models.

Given that there was no reduction due to bisimulation minimization for the `polling` instances (Table 5.1), no improvements in the running times can be expected for these models. Both other models, however, benefit from bisimulation minimization. For the larger instance of the `jobs` benchmark the regular verification runs into a shortage of memory, whereas the quotient model with a state space of about one fifth of the original model can be computed and verified in about 900 seconds. Both `mutex` instances can be solved in less time and without triggering an out-of-memory.

Before leaving the evaluation of the impact of bisimulation on model checking, we want to make two further observations. The first observation concerns the effectivity of bisimulation minimization. Apart from the avoided memouts, it paid off the most when the verification task is expensive and requires a lot of numerical operations. This is, for example, the case for the `embedded`, `coin` and `mutex` benchmarks. Unsurprisingly, the investment of the quotient construction is offset more easily in the face of expensive follow-up operations.

Secondly, we address a statement of [Wim10]:

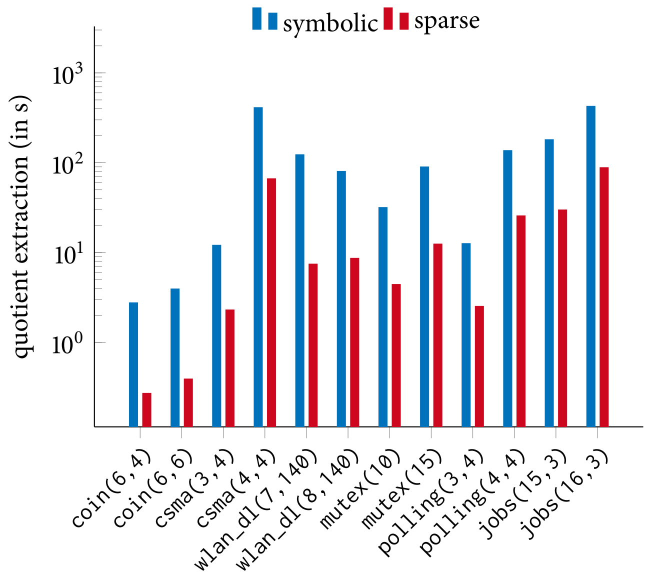


Figure 5.16: PA and MA benchmark models.

	model	instance	original verif.	sparse quotient constr.	sparse quotient verif.
MA	`mutex`	(10)	1734.59	76.57	245.10
		(15)	MO	186.00	1511.25
	`polling`	(3, 4)	183.79	18.36	186.80
		(4, 4)	MO	106.49	MO
	`jobs`	(15, 3)	189.15	298.34	77.09
		(16, 3)	MO	725.41	170.00

Table 5.4: Analysis of sparse quotient model checking.

»In summary, we can say that there is no reason to use floating point arithmetic for the computation of bisimulations for Markov chains. Rational arithmetic adds virtually no extra cost and yields a correct result that is not affected by rounding errors. «

In the cited source, performing bisimulation minimization using exact arithmetic was inspired by the observation that for some models the rounding problems led to increased quotient sizes or even nonterminating behavior. Ultimately, this is rooted in (i) the used DD libraries' (CUDD) setting that terminals that differ by at most a predefined constant ϵ are considered equal and stored only once and (ii) rounding errors through the use of floating point arithmetic itself. In practice, the former causes much more inacurracies in the context of bisimulation minimization than the latter. This is backed by our observation that for SYLVAN (that never treats distinct numbers as equal), the quotient sizes are the same for all considered models independent of whether floating point or rational arithmetic is used. Furthermore, we see that in our context, using rational arithmetic does not come for free. While we found that for the majority of experiments rational arithmetic has in fact negligible impact on the running times, for a few models it came at a considerable penalty. This is true, in particular, for the continuous-time models. The most extreme case is the `embedded(2000)` instance: computing the quotient using rational arithmetic took 50% more time than computing it using ordinary floating point arithmetic. Converting the model to a floating point representation prior to verification additionally invalidates most entries of the caches and causes a lot of new nodes to be inserted in the unique table (see Section 2.5), which in turn makes the verification slightly more expensive. This effect, however, vanishes over time as the conversion only happens once.

Reusing block numbers. Finally, we study the impact of reusing block numbers as suggested in [Dij16]. Figure 5.17 illustrates the effect of reusing blocks on the runtime of partition refinement (without quotient extraction) for selected models. Clearly, the impact is significant and, for example, reduces the computation time from more than 3600 seconds (timeout) to around 300 seconds for `jobs(15,3)`.

To study the effect more closely, we look at the progress of refinement over time for the `embedded(1000)` instance. Figure 5.18 shows the time spent on each iteration for this instance, where every iteration consists of computing the signature and then applying REFINE as in Algorithm 2. We observe that the time needed for REFINE is comparable in both settings. However, the time needed to compute the signature MTBDD grows rapidly when blocks are not reused. Ultimately, the major fraction of time is spent on signature computation. The figure also nicely illustrates that not reusing block numbers

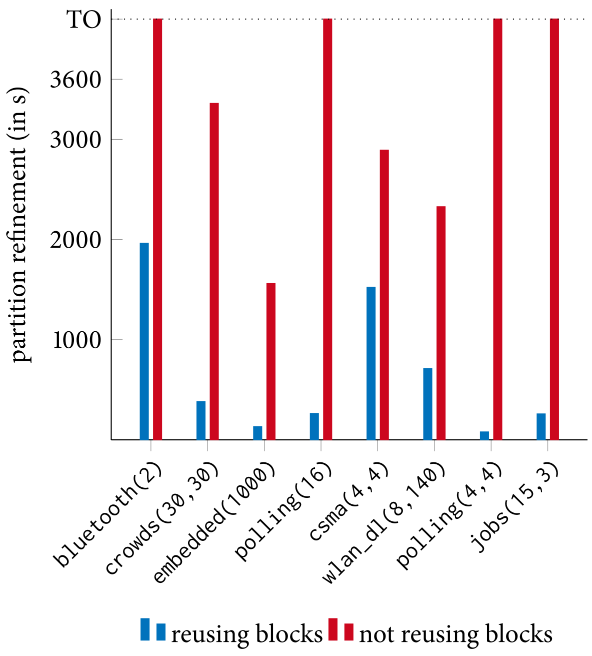


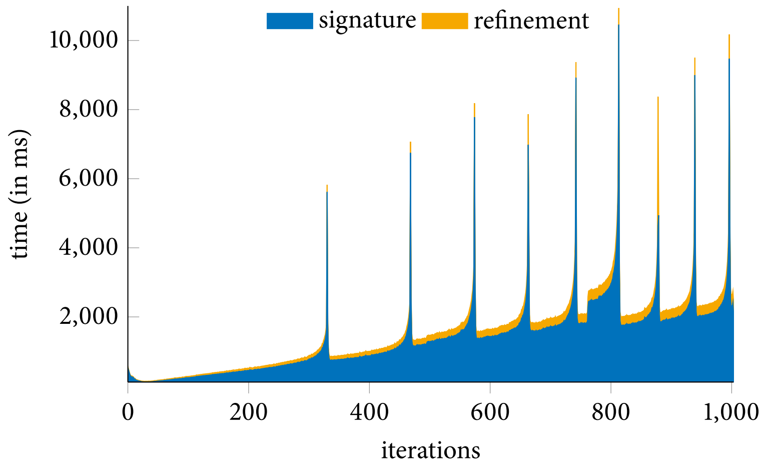
Figure 5.17: Influence of reusing blocks on partition refinement.

results in an increased number of nodes that are created. This can be seen by the periodic spikes that represent garbage collection. When reusing blocks, garbage collection is only triggered once at the very end. Before that, the time to compute the signature does not influence the overall time much.

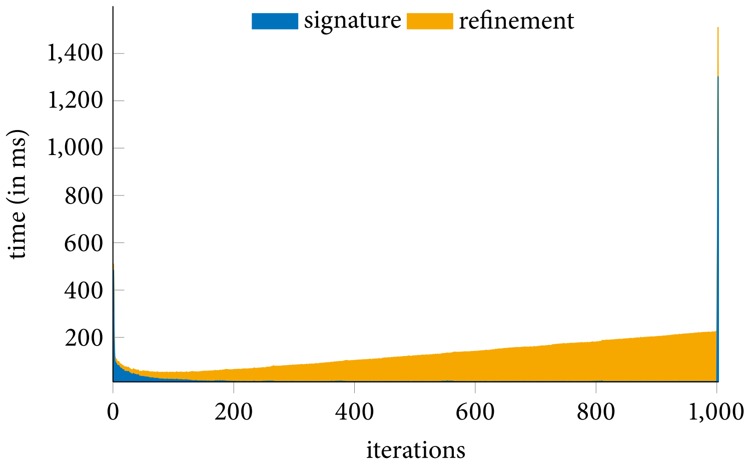
5.7 Conclusion and Future Work

In this chapter, we have shown how to minimize MRA represented as DDs with respect to strong bisimulation fully symbolically. We additionally showed how to extract the quotient in an explicit format to improve the runtimes of quotient extraction. Finally, we evaluated the effectiveness of bisimulation minimization in terms of regular quantitative model checking in the framework of Storm.

Our experiments show that performing bisimulation minimization does typically not



(a) Without reusing blocks.



(b) With reusing blocks.

Figure 5.18: Time spent on each refinement operation for `embedded(1000)`.

lead to reduced time and space requirements if the verification is performed using DDs. However, in the context of a hybrid verification approach that builds and minimizes the model symbolically and then solves the task on the explicit representation of the quotient, it can lead to substantial gains. More specifically, it enables the treatment of models that are otherwise out of reach for the hybrid approach.

So when is bisimulation minimization useful? Needless to say, models with a lot of symmetry benefit the most. However, it also crucially depends on the property to be verified. In general, we can say that it becomes more appealing the more expensive the verification task is. Such tasks tend to benefit the most, because the time spent on reducing the model is more easily outweighed by the potential gains. Therefore, bisimulation minimization is more likely to be applied when verifying complex tasks such as multi-objective [QJK17] or parametric [Deh+15] queries. These are currently only supported on explicit representations by probabilistic model checkers like Storm and Prism, because of their complexity and the involved numerical operations. As Storm supports the representation of parametric systems in terms of DDs, the existing implementations for explicit representations can be directly leveraged through the use of sparse quotient extraction. We want to remark that in the face of several, separate verification tasks, it crucially depends on the properties whether bisimulation minimization is fruitful. On the one hand, the minimization only has to be performed once and the minimized model can be used to answer all queries on a potentially smaller state space. However, in practice the initial partition is strongly influenced by the atomic propositions and reward models used in the properties and building a suitable quotient for several properties will therefore typically result in a much larger quotient model.

There are a number of directions to improve symbolic bisimulation minimization. For instance, it might be possible to apply the recent ideas in [GVV18] to reduce the amount of work in each refinement step. Another obvious dimension is the implementation and evaluation of weaker notions of bisimulation equivalence in the context of probabilistic model checking. However, the potential larger reductions come at the price of being computationally more involved. This seems to be promising in the context of continuous-time models, as they were the most resistant to strong bisimulation and the numerical queries tend to be expensive. Finally, bisimulation minimization may be fruitfully embedded in an abstraction-refinement loop. The (potentially) costly minimization could be interrupted and an over-approximation of the original system's behavior extracted. It is well conceivable that this already suffices to (dis)prove the validity of the property on the original model in some cases. If not, refinement may be continued until a conclusive answer can be given.

Chapter 6

Game-Based Abstraction-Refinement

6.1 Motivation and Goals

In Chapter 5, we argued that the state space explosion problem is one of the major obstacles for model checking and probabilistic model checking in particular. As state space sizes grow exponentially in the number of components and variables, representing the system in memory as well as the numerical solution methods become intractable in practice. Bisimulation minimization is a way to mitigate this explosion by identifying states of the system that behave equivalently and then merging these states to obtain a (mostly) smaller quotient system. As probabilistic bisimulation preserves PCTL* properties [Bai+05], properties can then be checked on the quotient model and the results carry over to the original model.

While this approach is able to significantly reduce the state spaces of models (see Section 5.6), it has two drawbacks. First, bisimulation minimization is mostly applicable to *finite* models. While there are approaches [DKP13] that can in principle treat infinite state spaces, they are restricted to cases where the bisimulation quotient is finite. After all, the quotient needs to be analyzed to obtain information about the original system. The second is that it is not property directed. The only influence the target property has on the quotient model is the selection of atomic propositions and whether or not to preserve rewards. Because of the preservation results for strong bisimulation (Theorem 1), the quotient model preserves *all* PCTL* and CSL properties, respectively [Bai+05]. Consequently, the bisimulation quotient may be much finer than necessary when only few properties are to be checked.

In this chapter, we present an abstraction technique for probabilistic automata for (unbounded) reachability objectives that is both property-driven and suitable for infinite-state systems. Similar to bisimulation minimization, it is based on grouping states and obtaining a "quotient model". However, the related states are not required to have equivalent behavior, but may – in general – behave differently. The quotient model therefore contains both the nondeterminism inherent to the model and the nondeterminism introduced by the abstraction. One way to resolve this is to merge the two sources of nondeterminism as in [DAr+01; DAr+02]. In this case, the quotient is again a probabilistic automaton that can be analyzed with standard techniques. As the abstract PA is an over-approximation of the original model, the minimal reachability probability in the quotient is a lower bound for the minimal reachability probability of the original model and, similarly, the maximal reachability probability in the quotient is an upper bound for the maximal reachability probability of the original model. Using this way of abstraction, probabilistic CEGAR [HWZ08; CV10] checks realizability of abstract counterexamples in the concrete model.

The alternative that we will focus on is to keep the sources of nondeterminism separate in the quotient model [KNP06a]. Instead of a PA, the abstraction then takes the form of a stochastic game (SG) [Sha53; Con90]. In this game, the players resolving the nondeterminism can either choose to cooperate or compete. The two layers of nondeterminism allow to maintain the direction (minimal or maximal) of the resolution of nondeterminism of the original model (induced by the property) while varying the direction of the abstraction. Ultimately, this results in lower and upper bounds for *both* minimal *and* maximal reachability probabilities in the original model. This allows for obtaining significantly tighter bounds than when the nondeterminism is simply merged.

Our presentation essentially summarizes and refines the results of [Wac11] and the associated publications [WZH07; WZ10; Hah+10b]. As such, there are strong similarities between our presentation and theirs and we will refrain from reciting the sources repeatedly. Similarly, the refinements we make to the aforementioned techniques were developed in the course of the Master thesis by Dimitri Bohlender [Boh14] (supervised by me) and this chapter reuses several examples from this work for illustration purposes.

For completeness, we mention other related techniques. *Magnifying-lens abstraction* [AR07] uses a similar scheme but considers individual concrete states contained in an abstract state and therefore "magnifies" the state. Even for infinite systems large parts of the probability mass may be concentrated in a finite subset of the states. *Sliding-window abstraction* [HMW09] is a technique to abstract from an infinite state space by "hiding" states that are irrelevant in the sense that they possess a negligible amount of probability mass in a way similar to the view through a window that slides as different states become more relevant over time. Finally, *assume-guarantee* reasoning [Kwi+10;

FKP10; KPC12] tries to combat the state space explosion problem by trying to analyze the components of a parallel composition in isolation.

6.2 Setting

Before discussing how to leverage stochastic games as abstractions of PA, we first fix the concrete setting. We are interested in minimal and maximal reachability probabilities in PA and for the rest of the chapter fix a (potentially infinite) PA $\mathcal{M} = \langle S, S^0, Act, \Delta \rangle$ and a set $T \subseteq S$ of target states. In our context, the PA $\mathcal{M}$ is given in terms of an SPA that is specified in the **JANI** language. For the rest of the chapter, let

$$\mathfrak{A} = \langle Loc, Var = PV \uplus TV, \nu_{TV}, TL, Act, \ell^0, Init^0, E \rangle$$

be an SPA without transient variables ($TV = \varnothing$) such that $\mathcal{M} = \langle S, S^0, Act, \Delta \rangle = [\![\mathfrak{A}]\!]$ as in Definition 27. In particular, recall that the state space of $\mathcal{M}$ is $S = Loc \times Val(Var)$ and that $\mathfrak{A}$ only has probabilistic edges. As $\mathcal{M}$ is given as a **JANI** model, the number of distributions $\Delta(s)$ available in each state $s \in S$ is finite. We assume that the target states T are given implicitly via an expression $\eta_T \in Bxp(Var)$ together with a location $\ell_T \in Loc$ such that

$$T = \{\langle \ell_T, \nu \rangle \in S \mid \nu \models \eta_T\}.$$

Without loss of generality, we assume that

» no state in $\mathcal{M}$ is a deadlock state, i. e. for all states $s \in S$ (also for unreachable states) there exists $\langle \alpha, \mu \rangle \in \Delta(s)$,

» $\mathfrak{A}$ is simple and therefore does not make use of indexed assignments, and

» all edges in $\mathfrak{A}$ are labeled with *different* actions $\alpha \in Act$.

To further ease the presentation, we require $\mathcal{M}$ to have one initial state, i. e., $S^0 = \{s^0\}$. Note that these restrictions can be lifted and do not affect the applicability of the approach.

However, there are two more serious restrictions that we need to make. One is that the probability expressions of the edges in $\mathfrak{A}$ do not contain any variables in *Var*. That is, for every location $\ell \in Loc$ and every edge $\langle g, \alpha, D \rangle \in E(\ell)$ we have

$$Var(D(a, \ell)) = \varnothing \text{ for all } \langle a, \ell \rangle \in supp(D).$$

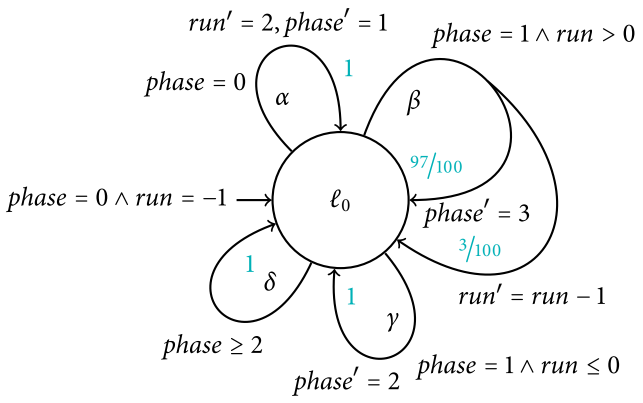


Figure 6.1: The SPA $\mathfrak{A}$ used as a running example.

The motivation behind this is simple: it is a syntactic restriction that guarantees that the abstractions of $\mathcal{M}$ we will discuss later are *finitely branching*. We therefore treat the expressions $D(a,\ell)$ of symbolic probability distributions as elements of the interval $[0,1]$. Note that this assumption also implies that there are only finitely many different probabilities appearing in $\mathcal{M}$, because the support sets of the symbolic distributions in $\mathfrak{A}$ are finite. The second significant restriction regarding the input model $\mathfrak{A}$ is that we require all expressions appearing in guards and assignments to belong to a *suitable theory whose satisfiability problem is decidable*. As it is arguably the most used in currently existing models, we assume that this theory is *linear integer arithmetic* (see Section 2.6).

Example 49. As a running example, we consider the SPA $\mathfrak{A}$ in Figure 6.1. It has one location ℓ_0 and two (permanent) variables *phase* and *run*. Where *phase* has the domain $\{0,1,2,3\}$ and is initially 0, $run \in \mathbb{Z}$ with initial value −1. Four actions $Act = \{\alpha,\beta,\gamma,\delta\}$ uniquely identify the four probabilistic edges $e_\alpha, e_\beta, e_\gamma$ and e_δ. Figure 6.2 shows the semantics of $\mathfrak{A}$ in terms of a PA $\mathcal{M} = [\![\mathfrak{A}]\!]$ where states are of the form $\langle phase, run\rangle$ and thus omitting the location component, because there is only a single location. For readability, we color the states according to their values of *phase* and label the distributions for future reference.

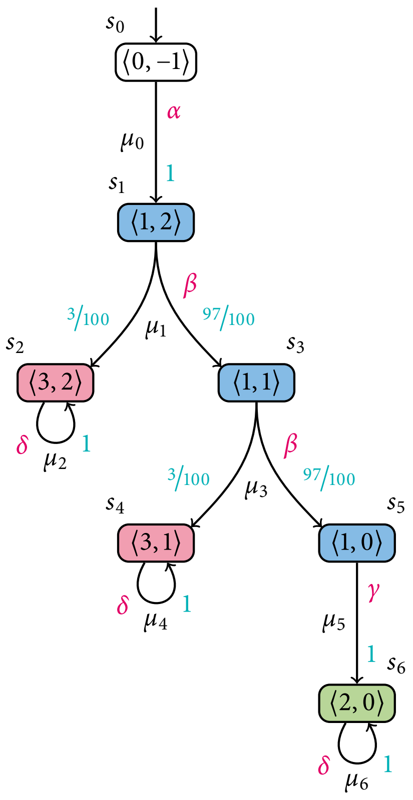


Figure 6.2: Reachable fragment of $\mathcal{M} = [\![\mathfrak{A}]\!]$ of the example SPA $\mathfrak{A}$.

Intuitively, $\mathfrak{A}$ behaves as follows. In the first step, the protocol is initialized and the counter *run* that indicates how many protocol iterations are remaining is set to 2. This counter is then decremented as long as it remains (strictly) positive. In each of these steps, the protocol fails (*phase* = 3) with probability $3/100$ and successfully decrements the counter with probability $97/100$. We are interested in the minimal probability that the system reaches a state with *phase* = 2 (and location ℓ_0), which indicates successful termination of the protocol.

6.3 Games as Abstractions

To further motivate the use of games as abstractions, we start our presentation by recapitulating the essentials of the abstract interpretation framework [CC77]. At its core, it uses the concept of a *Galois connection*.

Definition 39 (Galois Connection)**.** Let $\langle L, \sqsubseteq_L \rangle$ and $\langle M, \sqsubseteq_M \rangle$ be complete lattices. A pair $\langle \alpha, \gamma \rangle$ of monotonic functions

$$\alpha : L \to M \text{ and } \gamma : M \to L$$

is a *Galois connection* if

$$\forall \ell \in L \,.\, \ell \sqsubseteq_L \gamma(\alpha(\ell)) \text{ and } \forall m \in M \,.\, \alpha(\gamma(m)) \sqsubseteq_M m.$$

We refer to α and γ as the abstraction and concretization functions, respectively.

Intuitively, a Galois connection establishes a connection between a concrete domain L and an abstract domain M by providing functions that map concrete elements to abstract ones (via the *abstraction* α) and abstract ones to concrete ones (via the *concretization* γ). As the orderings $\sqsubseteq_L$ and $\sqsubseteq_M$ can be thought of as the degree of preciseness (with *smaller* elements being *more precise*), the conditions on the abstraction and concretization functions ensure that

(i) first abstracting a concrete element $\ell \in L$ and then concretizing the result cannot yield something more precise than ℓ, and

(ii) first concretizing an abstract element m and then abstracting the result does not lose information in the sense that it is at least as precise as m.

A popular technique for computing $\Pr^-_{\mathcal{M}}(\lozenge T)$ and $\Pr^+_{\mathcal{M}}(\lozenge T)$ for the PA $\mathcal{M}$ is *value iteration*, which is essentially a fixed-point analysis on the complete lattice $\left\langle [0,1]^S, \leq \right\rangle$. For $\sqsubseteq \in \{\leq, \geq\}$, we refer to $L_\sqsubseteq = \left\langle [0,1]^S, \sqsubseteq \right\rangle$ as the *concrete domains*. More specifically, value iteration approximates minimal and maximal reachability probabilities by iterating the (value iteration) value transformers

$$vi^-_{\mathcal{M},T}(w)(s) = \begin{cases} 1 & \text{if } s \in T \\ \displaystyle\min_{\langle \alpha, \mu \rangle \in \Delta(s)} \sum_{s' \in S} \mu(s') \cdot w(s') & \text{otherwise} \end{cases}$$

and

$$vi^{+}_{\mathcal{M},T}(w)(s) = \begin{cases} 1 & \text{if } s \in T \\ \max\limits_{\langle\alpha,\mu\rangle\in\Delta(s)} \sum\limits_{s'\in S} \mu(s')\cdot w(s') & \text{otherwise} \end{cases}$$

on valuations $w \in [0,1]^S$ starting from the constant zero valuation 0^S, which is the least element in $L_{\leq}$. To shorten the notation, we from now on use ☆ to range over $\{-,+\}$ and let ★ refer to min if ☆ $= -$ and max otherwise. For example, we write

$$vi^{☆}_{\mathcal{M},T}(w)(s) = \begin{cases} 1 & \text{if } s \in T \\ \underset{\langle\alpha,\mu\rangle\in\Delta(s)}{★} \sum\limits_{s'\in S} \mu(s')\cdot w(s') & \text{otherwise} \end{cases}$$

to define both transformers $vi^{-}_{\mathcal{M},T}$ and $vi^{+}_{\mathcal{M},T}$.

Since minimal and maximal reachability probabilities can be expressed in terms of the least-fixed points of these transformers as

$$\Pr^{☆}_{s}(\Diamond T) = \mathit{lfp}_{\leq}(vi^{☆}_{\mathcal{M},T})(s) \tag{6.1}$$

and the Kleene fixed-point theorem states that

$$\mathit{lfp}_{\leq}(vi^{☆}_{\mathcal{M},T}) = \sup_{n\in\mathbb{N}} \left\{ \left(vi^{☆}_{\mathcal{M},T}\right)^{n}(0^S) \right\}$$

the ascending chains resulting from iterating the value iteration transformers on 0^S converge to the minimal and maximal reachability probabilities, respectively.

Fixing a partition Π of the state space S with finitely many blocks for the remainder of the chapter, the natural *abstract domains* of computation are the complete lattices $M_{\sqsubseteq} = \langle [0,1]^{\Pi}, \sqsubseteq \rangle$ for $\sqsubseteq \in \{\leq, \geq\}$. [Wac11] shows that $\langle \alpha^l, \gamma \rangle$ and $\langle \alpha^u, \gamma \rangle$ with

$$\begin{aligned} \alpha^l(w)(B) &= \inf_{s\in B} w(s) \text{ for all } B \in \Pi \\ \alpha^u(w)(B) &= \sup_{s\in B} w(s) \text{ for all } B \in \Pi \\ \gamma(w^{\#})(s) &= w^{\#}(B(s)) \text{ for all } s \in S \end{aligned}$$

for $w \in [0,1]^S, w^{\#} \in [0,1]^{\Pi}$ are Galois connections: $\langle \alpha^l, \gamma \rangle$ is a Galois connection between $L_{\geq}$ and $M_{\geq}$ whereas $\langle \alpha^u, \gamma \rangle$ is a Galois connection between $L_{\leq}$ and $M_{\leq}$.

Using the connections between the concrete and abstract domains, we proceed to connect concrete and abstract valuation transformers. Let $f\colon [0,1]^S \to [0,1]^S$ be a concrete valuation transformer. An abstract transformer $g^{\#}\colon [0,1]^{\Pi} \to [0,1]^{\Pi}$ *soundly approximates* f in $M_{\sqsubseteq}$ if

$$f(\gamma(w^{\#})) \sqsubseteq \gamma(g^{\#}(w^{\#})). \tag{6.2}$$

Intuitively, this property states that the applying the transformation f in the concrete domain must be at least as precise as the abstract transformation in the abstract domain. There may be several abstract transformers that satisfy this property. However, in general, for non-trivial partitions Π, abstract transformers are not as precise as their concrete counterparts. [CC92] shows that the most precise abstract transformer $f^{\#}$ that is sound with respect to (6.2) is given by

$$f^{\#} = \alpha \circ f \circ \gamma.$$

Therefore, $f^{\#}$ is often referred to as the *best transformer* with respect to the Galois connection $\langle \alpha, \gamma \rangle$. Using results from [CC92], [Wac11] shows that for $f_l^{\#}$ and $f_u^{\#}$ given by

$$f_l^{\#} = \alpha_l \circ f \circ \gamma \text{ and } f_u^{\#} = \alpha_u \circ f \circ \gamma$$

it is

$$\gamma(lfp_{\leq}(f_l^{\#})) \leq lfp_{\leq}(f) \leq \gamma(lfp_{\leq}(f_u^{\#})). \tag{6.3}$$

Together with Equation (6.1), this yields lower and upper bounds for the concrete reachability probabilities:

$$\gamma(lfp_{\leq}(vi^{\#,\star,l}_{\mathcal{M},T}))(s) \leq \mathrm{Pr}^{\star}_s(\Diamond T) \leq \gamma(lfp_{\leq}(vi^{\#,\star,u}_{\mathcal{M},T}))(s)$$

where

$$vi^{\#,\star,l}_{\mathcal{M},T} = \alpha_l \circ vi^{\star}_{\mathcal{M},T} \circ \gamma \tag{6.4}$$

$$vi^{\#,\star,u}_{\mathcal{M},T} = \alpha_u \circ vi^{\star}_{\mathcal{M},T} \circ \gamma. \tag{6.5}$$

Given this connection, we now show that these abstract transformers can be captured in the form of stochastic games.

Definition 40 (Stochastic Game)**.** A *stochastic game (SG)* is a tuple

$$\mathcal{G} = \left\langle \mathcal{V} = \mathcal{V}_1 \uplus \mathcal{V}_2 \uplus \mathcal{V}_p, \mathcal{E}, \mathcal{V}_0 \right\rangle$$

where

- » $\langle \mathcal{V}, \mathcal{E} \rangle$ is a directed graph with (finite) vertex set $\mathcal{V}$ and (finite) edge set $\mathcal{E} \subseteq (\mathcal{V}_1 \times \mathcal{V}_2) \cup (\mathcal{V}_2 \times \mathcal{V}_p) \cup (\mathcal{V}_p \times \mathcal{V}_1)$,
- » $\mathcal{V}_1$ is the set of player 1 vertices,
- » $\mathcal{V}_2$ is the set of player 2 vertices,
- » $\mathcal{V}_p \subseteq \mathit{Dist}(\mathcal{V}_1)$ is the set of probabilistic vertices where for all $\mu \in \mathcal{V}_p$ we have $\mu(v') > 0$ if and only if $\langle \mu, v' \rangle \in \mathcal{E}$, and
- » $\mathcal{V}_0 \subseteq \mathcal{V}_1$ are the initial vertices.

In an SG, every vertex belongs to either player 1 or 2 or is a probabilistic vertex, which is a probability distribution over player 1 vertices. The two players resolve the nondeterminism in the vertices that belong to them. That is, in player 1 vertices, player 1 makes a decision which player 2 successor to pick and in player 2 vertices it is player 2's turn to choose a successor (probabilistic) vertex. Note that the edge relation of our definition of an SG ensures that the game is played in a strictly alternating manner: first, player 1 makes a choice, then player 2 and finally the next player 1 vertex is chosen according to a probability distribution. We denote the successors and predecessors of a vertex $v \in \mathcal{V}$ by $\mathit{succ}_{\mathcal{G}}(v) = \{v' \in \mathcal{V} \mid \langle v, v' \rangle \in \mathcal{E}\}$ and $\mathit{pred}_{\mathcal{G}}(v) = \{v' \in \mathcal{V} \mid \langle v', v \rangle \in \mathcal{E}\}$ and omit the subscript $\mathcal{G}$ if it clear from the context.

Definition 41 (Strategies for Stochastic Game)**.** Let $\mathcal{G} = \left\langle \mathcal{V} = \mathcal{V}_1 \uplus \mathcal{V}_2 \uplus \mathcal{V}_p, \mathcal{E}, \mathcal{V}_0 \right\rangle$ be an SG. A (memoryless, deterministic) strategy σ_i for player $i \in \{1, 2\}$ in $\mathcal{G}$ is a function

$$\sigma_i : \mathcal{V}_i \to \mathcal{V}$$

such that $\sigma_i(v_i) \in \mathit{succ}(v_i)$. We denote the set of player i strategies for $\mathcal{G}$ by $\mathfrak{S}_i^{\mathcal{G}}$.

A (memoryless, deterministic) strategy pair $\langle \sigma_1, \sigma_2 \rangle$ for $\mathcal{G}$ consists of a player 1 strategy σ_1 and a player 2 strategy σ_2.

Intuitively, a strategy pair $\langle \sigma_1, \sigma_2 \rangle$ and an SG $\mathcal{G}$ induce a (finite) DTMC $\mathcal{G}^{\langle \sigma_1, \sigma_2 \rangle}$ by resolving all nondeterminism in $\mathcal{G}$. Using the standard unique probability measure $\mathrm{Pr}_{\mathcal{G}}^{\langle \sigma_1, \sigma_2 \rangle}$ on $\mathcal{G}^{\langle \sigma_1, \sigma_2 \rangle}$, we define

$$\mathrm{Pr}_{\mathcal{G}}^{\star_1 \star_2}(\Diamond T) = \bigstar_{1\,\sigma_1 \in \mathfrak{S}_1^{\mathcal{G}}} \; \bigstar_{2\,\sigma_2 \in \mathfrak{S}_2^{\mathcal{G}}} \; \mathrm{Pr}_{\mathcal{G}}^{\langle \sigma_1, \sigma_2 \rangle}(\Diamond T).$$

When both players cooperate, reachability probabilities can be computed as for PA. However, in our context, the case when both players pursue *competing goals* is of particular relevance. Similar to PA, one can show that these extremal reachability probabilities are captured by the least fixed-point of particular transformers. For this, we define four transformers as follows:

$$vi_{\mathcal{G},T}^{\star_1 \star_2}(w)(v) = \begin{cases} 1 & \text{if } v \in T \\ \bigstar_{1\,v_2 \in succ(v)} \; \bigstar_{2\,v_p \in succ(v_2)} \; \sum_{v' \in \mathcal{V}} v_p(v') \cdot w(v') & \text{otherwise.} \end{cases}$$

We can then [Con92; Wac11] use

$$\mathrm{Pr}_{\mathcal{G}}^{\star_1 \star_2}(\Diamond T) = \mathit{lfp}_{\leq}(vi_{\mathcal{G},T}^{\star_1 \star_2})$$

to characterize reachability probabilities in SG. Similar to the PA case, *value iteration* approximates the probabilities by iterating these transformers starting from $0^{\mathcal{V}_1}$.

To ease the presentation in the rest of the chapter, we introduce further notation. For a given PA $\mathcal{M} = \left\langle S, S^0, Act, \Delta \right\rangle$ and a partition Π of S, we use

$$[\Delta(s, \alpha)]_{\Pi} = \{[\mu]_{\Pi} \mid \langle s, \alpha, \mu \rangle \in \Delta\}$$

to denote all distributions labeled with α that are available in state $s \in S$ lifted to the partition Π. We extend this notation by

$$[\Delta(s)]_{\Pi} = \bigcup_{\alpha \in Act(s)} [\Delta(s, \alpha)]_{\Pi} \text{ and } [\Delta]_{\Pi} = \bigcup_{s \in S} [\Delta(s)]_{\Pi}$$

to additionally abstract from the action and the concrete state. We remark that in our context (see Section 6.2), $[\Delta(s, \alpha)]_{\Pi}$, $[\Delta(s)]_{\Pi}$ and $[\Delta]_{\Pi}$ are all finite if the partition has finitely many blocks.

6.3.1 Game-based Abstraction

From now on, we assume $T \subseteq S$ to be *exactly representable* in the partition Π of S, i. e.

$$\exists T^{\#} \subseteq \Pi \,.\, T = \bigcup_{B \in T^{\#}} B.$$

and use $T^{\#}$ to refer to the blocks of Π that exactly represent T.

Using the value iteration transformers $vi^{\star}_{\mathcal{M},T}$ on PA and the best transformers in Equations (6.4) and (6.5), we obtain

$$vi^{\#,\star,l}_{\mathcal{M},T}(w^{\#})(B) = \begin{cases} 1 & \text{if } B \in T^{\#} \\ \min\limits_{s \in B} \ \bigstar\limits_{\langle \alpha,\mu \rangle \in \Delta(s)} \ \sum\limits_{s' \in S} \mu(s') \cdot w^{\#}(B) & \text{otherwise} \end{cases}$$

$$vi^{\#,\star,u}_{\mathcal{M},T}(w^{\#})(B) = \begin{cases} 1 & \text{if } B \in T^{\#} \\ \max\limits_{s \in B} \ \bigstar\limits_{\langle \alpha,\mu \rangle \in \Delta(s)} \ \sum\limits_{s' \in S} \mu(s') \cdot w^{\#}(B) & \text{otherwise.} \end{cases}$$

Here, we can move from the infimum and supremum to taking the minimum and maximum, respectively, by exploiting that

» Π has finitely many blocks,

» $\mathcal{M}$ is finitely branching, and

» there are only finitely many probabilities appearing in $\mathcal{M}$ (see Section 6.2).

From the structure of these equations, the similarity to the value iteration transformers $vi^{\star_1 \star_2}_{\mathcal{G},T}$ for games is apparent. We now introduce the *game-based* abstraction [Kat+10] of a PA $\mathcal{M}$ with respect to Π as an SG $\mathcal{G}^{\text{gba}}_{\mathcal{M},\Pi}$ in such a way that the value iteration transformers on $\mathcal{G}^{\text{gba}}_{\mathcal{M},\Pi}$ coincide with the abstract value iteration transformers on $\mathcal{M}$.

Definition 42 (Game-based Abstraction)**.** The *game-based abstraction* of $\mathcal{M}$ with respect to Π is the SG

$$\mathcal{G}^{\text{gba}}_{\mathcal{M},\Pi} = \left\langle \mathcal{V} = \mathcal{V}_1 \uplus \mathcal{V}_2 \uplus \mathcal{V}_p, \mathcal{E}, \mathcal{V}_0 \right\rangle$$

where

» the player 1 vertices $\mathcal{V}_1 = S/\Pi$ are the blocks of Π,

» the probabilistic vertices $\mathcal{V}_p = [\Delta]_\Pi$ are the (lifted) distributions occurring in $\mathcal{M}$,

» the player 2 vertices $\mathcal{V}_2 = \{[\Delta(s)]_\Pi \mid s \in S\}$ are sets of probabilistic vertices,

» the initial vertices $\mathcal{V}_0 = S^0/\Pi$

and the edge relation $\mathcal{E}$ is given as

$$\begin{aligned}\mathcal{E} = &\left\{\left\langle v_p, v_1\right\rangle \in \mathcal{V}_p \times \mathcal{V}_1 \mid v_p(v_1) > 0\right\}\\ &\cup \left\{\left\langle v_2, v_p\right\rangle \in \mathcal{V}_2 \times \mathcal{V}_p \mid v_p \in v_2\right\}\\ &\cup \left\{\langle v_1, v_2\rangle \in \mathcal{V}_1 \times \mathcal{V}_2 \mid \exists s \in v_1 \,.\, v_2 = [\Delta(s)]_\Pi\right\}\end{aligned}$$

Intuitively, the player 1 vertices comprise all blocks of the partition. In such a vertex, player 1 chooses a state $s \in S$ among all concrete states contained in the block. Player 2 can then choose a (lifted) probability distribution available in s that then probabilistically determines the next player 1 state.

Example 50. Reconsider the running example SPA $\mathfrak{A}$ from Example 49. Let the partition Π be given as

$$\Pi = \left\{ \underbrace{\{s_0\}}_{B_0}, \underbrace{\{s_1, s_3, s_5\}}_{B_1}, \underbrace{\{s_2, s_4\}}_{B_2}, \underbrace{\{s_6\}}_{B_3} \right\}$$

and note that our target set $T = \{6\}$ is trivially exactly representable in Π. Figure 6.3 shows the game-based abstraction $\mathcal{G}^{\text{gba}}_{\mathcal{M},\Pi}$ where we draw player 2 vertices as squares and probabilistic vertices as black dots. The most interesting block is B_1 that subsumes three states. As $s_1 \in B_1$, there is a player 2 successor of B_1, namely v_2^1, that enables all lifted distributions available in s_1. In our example, this is $[\mu_1]_\Pi$, which assigns probability $3/100$ to B_2 and $97/100$ to B_1. The latter loop is the result of treating s_3 as being equivalent to s_1 by the equivalence relation associated with Π.

It is not difficult to see that applying the value iteration transformers $vi^{-\star_2}_{\mathcal{G}^{\text{gba}}_{\mathcal{M},\Pi},T^\#}$ and $vi^{+\star_2}_{\mathcal{G}^{\text{gba}}_{\mathcal{M},\Pi},T^\#}$ yields the desired abstract transformers $vi^{\#,\star_2,l}_{\mathcal{M},T}(w^\#)$ and $vi^{\#,\star_2,u}_{\mathcal{M},T}(w^\#)$, respec-

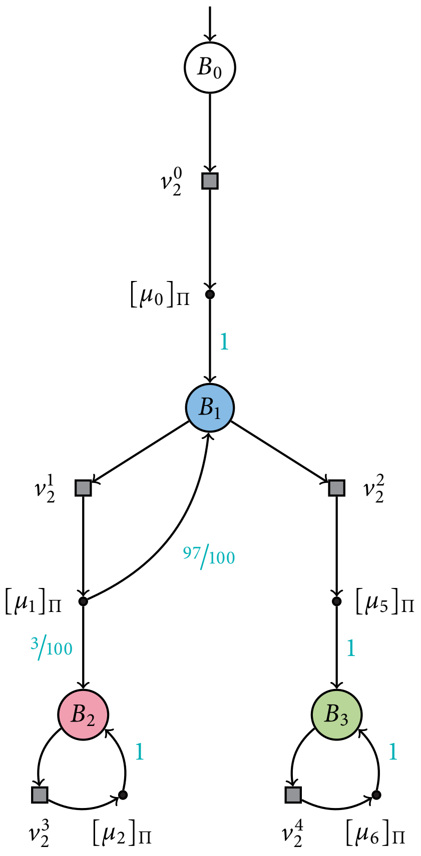


Figure 6.3: The game-based abstraction $\mathcal{G}^{\text{gba}}_{\mathcal{M},\Pi}$ for Example 50.

tively. This immediately gives rise to the following theorem.

Theorem 2 (Correctness of Game-based Abstraction [Kat+10])**.** *For* $\mathcal{G} = \mathcal{G}^{\text{gba}}_{\mathcal{M},\Pi}$

$$\mathrm{Pr}_{\mathcal{G}}^{-\star_2}(\Diamond T^{\#}) \leq \mathrm{Pr}_{\mathcal{M}}^{\star_2}(\Diamond T) \leq \mathrm{Pr}_{\mathcal{G}}^{+\star_2}(\Diamond T^{\#}).$$

This theorem asserts that the game-based abstraction from $\mathcal{M}$ and Π yields lower and upper bounds on both the minimal and maximal reachability probabilities.

Example 51. Recall the SG from Example 50. We have

$$\mathrm{Pr}_{\mathcal{G}}^{--}(\Diamond T^{\#}) = \mathrm{Pr}_{\mathcal{G}}^{-+}(\Diamond T^{\#}) = 0$$
$$\mathrm{Pr}_{\mathcal{G}}^{+-}(\Diamond T^{\#}) = \mathrm{Pr}_{\mathcal{G}}^{++}(\Diamond T^{\#}) = 1$$

and therefore only trivial bounds for the reachability probabilities in $\mathcal{M}$ can be derived from the game-based abstraction for the selected partition Π.

While game-based abstraction refinement has been shown to reduce model sizes by orders of magnitude [Kat+10], the same source observes that the construction of the game typically dominates the presented abstraction-refinement loop significantly. Recall that our ultimate goal is to build the abstraction from the symbolic representation of the PA in terms of an SPA directly without first building a representation of the concrete state space. While it is possible to derive $\mathcal{G}^{\text{gba}}_{\mathcal{M},\Pi}$ from such models using Smt solvers, it inherently requires to consider *combinations of edges* in the abstraction process. Ultimately, this is rooted in the fact that the value iteration transformers for the PA and the associated best abstract transformers optimize over all distributions available in a specific state, which arise from different edges of the symbolic model. As mentioned above, this can be prohibitively expensive for the abstraction process. Optimally, concerning the abstraction process, we would like to consider all edges *individually* and combine the abstractions to obtain an abstraction of the overall model.

6.3.2 Menu-based Abstraction

Consider the concrete transformer

$$me^{\star}_{\mathcal{M},T}(w)(s) = \mathop{\bigstar}_{\alpha \in Act(s)} me^{\star}_{\mathcal{M},T}[\alpha](w)(s)$$

that updates the value of a state by optimizing over the sub-transformer $me^{\star}_{\mathcal{M},T}[\alpha]$. For the case where $\star = +$, the sub-transformer is given by

$$me^{+}_{\mathcal{M},T}[\alpha](w)(s) = \begin{cases} 1 & \text{if } s \in T \\ 0 & \text{if } s \notin T \wedge \alpha \notin Act(s) \\ \max\limits_{\langle \alpha,\mu\rangle \in \Delta(s)} \sum\limits_{s' \in S} \mu(s') \cdot w(s') & \text{otherwise.} \end{cases}$$

We call these transformers *menu-based (sub)transformers* as the overall transformer $me^{\star}_{\mathcal{M},T}$ selects the action "from a menu". The maximizing menu-based subtransformer first maximizes over all choices of a given state s with a particular action label α and then maximizes over all available actions in s to obtain the value of s in the transformed valuation. As it turns out, we have to be bit more careful for the minimizing counterpart $me^{-}_{\mathcal{M},T}[\alpha]$ and define

$$me^{-}_{\mathcal{M},T}[\alpha](w)(s) = \begin{cases} 1 & \text{if } s \in T \vee \alpha \notin Act(s) \\ \min\limits_{\langle \alpha,\mu\rangle \in \Delta(s)} \sum\limits_{s' \in S} \mu(s') \cdot w(s') & \text{otherwise.} \end{cases}$$

The difference here is that the transformer also needs to return 1 in case the action is not enabled in the selected state. Intuitively, this is because it is implicitly forbidden to select this action in the state if it is not even enabled.

Clearly, we have $vi^{\star}_{\mathcal{M},T} = me^{\star}_{\mathcal{M},T}$, so until now we have only rephrased the transformer definition. However, the chosen formulation allows to consider the subtransformer $me^{+}_{\mathcal{M},T}[\alpha]$ separately. Letting

$$\begin{aligned} me^{\#,\star,l}_{\mathcal{M},T}[\alpha] &= \alpha_l \circ me^{\#,\star}_{\mathcal{M},T}[\alpha] \circ \gamma, \text{and} \\ me^{\#,\star,u}_{\mathcal{M},T}[\alpha] &= \alpha_u \circ me^{\#,\star}_{\mathcal{M},T}[\alpha] \circ \gamma \end{aligned}$$

be the best abstract transformers for the menu-based subtransformer with respect to the lower and upper abstractions of the two Galois connections, we can define the abstract transformers as

$$\begin{aligned} me^{\#,\star,l}_{\mathcal{M},T}(w^{\#})(B) &= \min_{\alpha \in Act(B)} me^{\#,\star,l}_{\mathcal{M},T}[\alpha](w^{\#})(B) \\ me^{\#,\star,u}_{\mathcal{M},T}(w^{\#})(B) &= \max_{\alpha \in Act(B)} me^{\#,\star,u}_{\mathcal{M},T}[\alpha](w^{\#})(B). \end{aligned}$$

Similar to the value iteration transformers, we have

$$\gamma(lfp_{\leq}(me^{\#,\star,l}_{\mathcal{M},T}))(s) \leq \Pr^{\star}_{s}(\lozenge T) \leq \gamma(lfp_{\leq}(me^{\#,\star,u}_{\mathcal{M},T}))(s)$$

due to Equation (6.3). Again, we can formulate an SG such that the value iteration transformers on the game correspond to the abstract menu-based transformers.

Definition 43 (Menu-based Abstraction)**.** Let $\mathcal{M} = \langle S, S^0, Act, \Delta \rangle$ be a PA and Π a partition of S. The *menu-based abstraction* (or simply *menu-game*) of $\mathcal{M}$ with respect to Π is the SG

$$\mathcal{G}^{\mathrm{mba}}_{\mathcal{M},\Pi} = \left\langle \mathcal{V} = \mathcal{V}_1 \uplus \mathcal{V}_2 \uplus \mathcal{V}_p, \mathcal{E}, \mathcal{V}_0 \right\rangle$$

where

» the player 1 vertices $\mathcal{V}_1 = S/\Pi \uplus \{v^1_\perp\}$ are the blocks of Π together with a unique trap vertex $v^1_\perp$,

» the player 2 vertices $\mathcal{V}_2 = \{\langle v_1, \alpha\rangle \in \mathcal{V}_1 \times Act \mid \exists \alpha \in Act(v_1)\} \uplus \{v^2_\perp\}$ are induced by the actions enabled in the states of the PA that are contained in the player 1 vertices,

» the probabilistic vertices $\mathcal{V}_p = [\Delta]_\Pi \uplus \{v^p_\perp\}$ are the (lifted) distributions occurring in $\mathcal{M}$,

» the initial vertices $\mathcal{V}_0 = S^0/\Pi$

and the edge relation $\mathcal{E}$ is given as

$$\begin{aligned}
\mathcal{E} = &\{\langle v_1, v_2\rangle \in \mathcal{V}_1 \times \mathcal{V}_2 \mid v_2 = \langle v_1, \alpha\rangle \wedge \alpha \in Act(v_1)\} \\
&\cup \left\{\langle v_2, v_p\rangle \in \mathcal{V}_2 \times \mathcal{V}_p \mid v_2 = \langle v_1, \alpha\rangle \wedge \exists s \in v_1 \,.\, v_p \in [\Delta(s,\alpha)]_\Pi\right\} \\
&\cup \left\{\langle v_2, v^p_\perp\rangle \in \mathcal{V}_2 \times \mathcal{V}_p \mid v_2 = \langle v_1, \alpha\rangle \wedge \exists s \in v_1 \,.\, \alpha \notin Act(s)\right\} \\
&\cup \left\{\langle v^1_\perp, v^2_\perp\rangle, \langle v^2_\perp, v^p_\perp\rangle\right\} \\
&\cup \left\{\langle v_p, v_1\rangle \in \mathcal{V}_p \times \mathcal{V}_1 \mid v_p(v_1) > 0\right\}.
\end{aligned}$$

Example 52. Reconsider the running example SPA $\mathfrak{A}$ from Example 49 and the partition Π from Example 50. The menu-based abstraction $\mathcal{G}^{\mathrm{mba}}_{\mathcal{M},\Pi}$ is shown in Figure 6.4. Again, the most interesting block is B_1. Since B_1 subsumes a state (s_1) for which γ is not enabled as well as a state (s_5) for which β is not enabled, the two player 2 successors of B_1 both have the option to move to the (probabilistic) bottom state

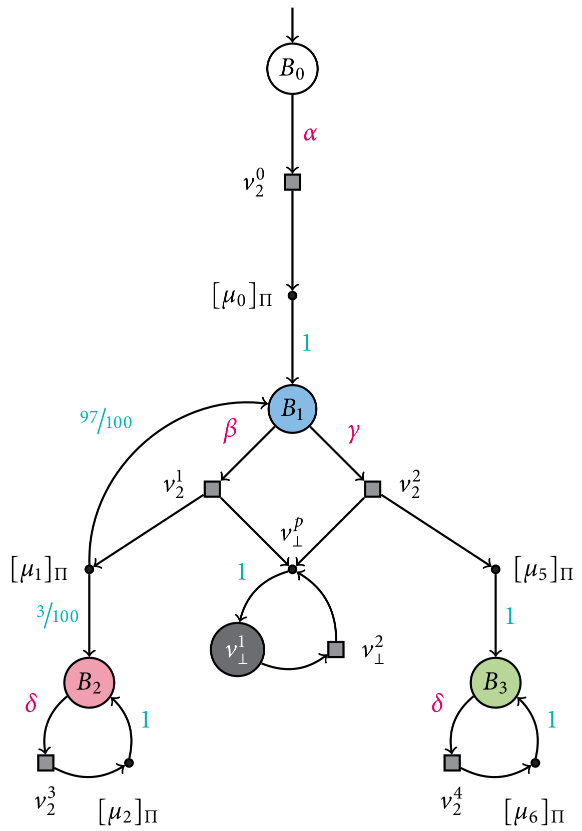


Figure 6.4: The menu-based abstraction $\mathcal{G}^{\text{mba}}_{\mathcal{M},\Pi}$ for Example 52.

$v_\perp^p$. The other choices of these player 2 vertices are the same as for the game-based abstraction and therefore correspond to $[\mu_1]_\Pi$ and $[\mu_5]_\Pi$, respectively.

We remark that this definition becomes slightly more complex in the presence of deadlock states in $\mathcal{M}$, which we ruled out by our assumptions (Section 6.2). Intuitively, this is because they have no possible way to continue, but the abstract game does not reflect that. That is, deadlock states are simply disregarded, which is clearly wrong. To account for deadlock states, the blocks that contain such states can, for example, be equipped with a self-loop.

One can show that the value iteration transformers on $\mathcal{G}^{\text{mba}}_{\mathcal{M},\Pi}$ correspond to the abstract menu-based transformers in the following sense:

$$\begin{aligned}
vi^{--}_{\mathcal{G}^{\text{mba}}_{\mathcal{M},\Pi},T^\# \cup \{v_\perp^1\}} &= me^{\#,-,l}_{\mathcal{M},T} \\
vi^{-+}_{\mathcal{G}^{\text{mba}}_{\mathcal{M},\Pi},T^\# \cup \{v_\perp^1\}} &= me^{\#,-,u}_{\mathcal{M},T} \\
vi^{+-}_{\mathcal{G}^{\text{mba}}_{\mathcal{M},\Pi},T^\#} &= me^{\#,+,l}_{\mathcal{M},T} \\
vi^{++}_{\mathcal{G}^{\text{mba}}_{\mathcal{M},\Pi},T^\#} &= me^{\#,+,u}_{\mathcal{M},T}.
\end{aligned}$$

Note that to realize the minimizing abstract menu-based transformers $me^{\#,-,l}_{\mathcal{M},T}$ and $me^{\#,-,u}_{\mathcal{M},T}$, the trap vertex $v_\perp^1$ is considered as an additional target state. This is due to the idiosyncrasy of the $me^-_{\mathcal{M},T}[\alpha]$ transformer that returns 1 for states that do not have α enabled.

We can now summarize the correctness of menu-based abstraction as follows.

Theorem 3 (Correctness of Menu-based Abstraction). *For* $\mathcal{G} = \mathcal{G}^{\text{mba}}_{\mathcal{M},\Pi}$

$$\begin{aligned}
\Pr^{--}_{\mathcal{G}}\left(\Diamond\left(T^\# \cup \{v_\perp^1\}\right)\right) &\leq \Pr^-_{\mathcal{M}}(\Diamond T) \leq \Pr^{-+}_{\mathcal{G}}\left(\Diamond\left(T^\# \cup \{v_\perp^1\}\right)\right), \textit{ and} \\
\Pr^{+-}_{\mathcal{G}}\left(\Diamond T^\#\right) &\leq \Pr^+_{\mathcal{M}}(\Diamond T) \leq \Pr^{++}_{\mathcal{G}}\left(\Diamond T^\#\right).
\end{aligned}$$

It is clear that menu-based abstraction is at most as precise as game-based abstraction, because the latter is a representation of the best abstract transformers $vi^{\#,\star_2,l}_{\mathcal{M},T}(w^\#)$ and $vi^{\#,\star_2,u}_{\mathcal{M},T}(w^\#)$. [Wac11] shows that in general the menu-based abstraction may be less precise for a given partition Π. In other words, it might be that the bounds obtained from the menu-game provide less information than the bounds obtained from the game-based abstraction. While losing precision is clearly not desirable, we want to stress that the menu-game can be derived by considering edges of the SPA in isolation. Because of this appealing property, we from now on focus solely on them.

Example 53. As the bounds obtained from the menu-game in Example 52 can be at most as precise as the ones obtained from the game-based abstraction in Example 50, we trivially also obtain the bounds $[0,1]$ from the former for both minimal and maximal reachability probabilities with respect to T.

6.3.3 Representing Menu-Games using Decision Diagrams

We now describe how to represent menu-games using DDs. Similar to representing Markov chains, probabilistic automata and Markov automata (see Section 2.5), we use the variable set

$$Var_{\mathcal{G}} = \underbrace{\{x_1,\ldots,x_n\}}_{\mathcal{S}} \uplus \underbrace{\{x_1',\ldots,x_n'\}}_{\mathcal{S}'} \uplus \underbrace{\{\mathfrak{a}_1,\ldots,\mathfrak{a}_k\}}_{\mathcal{A}} \uplus \underbrace{\{o_1,\ldots,o_m\}}_{\mathcal{O}}.$$

Here, we use

» $\mathcal{S}$ and $\mathcal{S}'$ encode the player 1 source and target vertices including $v_\perp^1$,

» the k variables in $\mathcal{A}$ to encode the choices of player 1, and

» the m variables in $\mathcal{O}$ to encode the choices of player 2.

Because of the strictly alternating fashion in which players take turns in our notion of stochastic games, we can encode player 2 vertices over the variables $\mathcal{S} \uplus \mathcal{A}$ and probabilistic vertices over $\mathcal{S} \uplus \mathcal{A} \uplus \mathcal{O}$. To do this, we also assume that player 2 and probabilistic vertices are not shared between different player 1 and player 2 vertices. That is, for $v_2 \in \mathcal{V}_2$ and $v_p \in \mathcal{V}_p$, $pred(v_2)$ and $pred(v_p)$ are singletons. While the former is already implied by Definition 43, the latter can be achieved by creating copies of probabilistic vertices without affecting the reachability probabilities of the game. Because of this uniqueness, we sometimes qualify probabilistic vertices v_p by their corresponding player 2 vertex $\langle v_1, \alpha\rangle$ and write $\langle v_1, \alpha, v_p\rangle$ instead of just v_p. Similar to before, we assume encodings $\langle v_1\rangle$, $\langle\langle v_1, \alpha\rangle\rangle$, $\langle\langle v_1, \alpha, \mu\rangle\rangle$ for player 1, 2 and probabilistic vertices, respectively, We then encode the edge relation $\mathcal{E}$ enriched with the distributions given by the probabilistic vertices as an MTBDD $\mathsf{M}_{\mathcal{E}}$ such that

$$\mathsf{M}_{\mathcal{E}}(\mathcal{S} \leftarrow \langle v\rangle, \mathcal{A} \leftarrow \langle\langle v,\alpha\rangle\rangle, \mathcal{O} \leftarrow \langle\langle v,\alpha,\mu\rangle\rangle, \mathcal{S}' \leftarrow \langle v'\rangle)$$
$$= \begin{cases} \mu(v') & \text{if } \alpha \in Act(v) \wedge \exists s \in v\,.\, \left(\mu \in [\Delta(s,\alpha)]_\Pi \vee \left(\alpha \notin Act(s) \wedge \mu = \delta_{v_\perp^1}\right)\right) \\ 0 & \text{otherwise.} \end{cases}$$

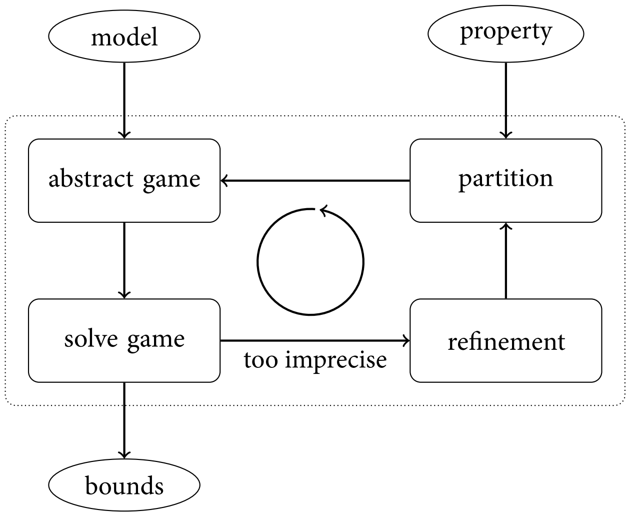


Figure 6.5: Overview of Abstraction-Refinement using Games.

6.4 Abstraction-Refinement using Games

In this section, we formulate an abstraction-refinement loop. For the abstraction part, we use the aforementioned menu-games as a way to implement the abstract menu-based transformers. Figure 6.5 gives an overview of the approach. First, the abstract game is built from the current partition and the symbolic model description, which in our case is the SPA $\mathfrak{A}$. Initially, the partition is derived from the target set T given by the property. Solving the game with proper optimization directions for the two players then yields lower and upper bounds on the minimal and maximal probabilities to satisfy the reachability property. If the bounds are precise enough, they are returned as the analysis result. In case the bounds are too imprecise, a refinement needs to take place that results in a finer partition. This loop is iterated until a desired precision is achieved or a given verification task can be conclusively answered.

6.4.1 Predicate Abstraction

Both game-based abstractions have in common that they “merge” states in the form of player 1 vertices. In our context, we strive to derive the abstract games directly from $\mathfrak{A}$ rather than building $\mathcal{M}$ first and then extracting the game from it. To this end, we employ *predicate abstraction* [GS97]. Here, we assume that the partition Π of the state space of $\mathcal{M}$ is given implicitly by a set of predicates $\Phi \subseteq \mathit{Bxp}(\mathit{Var})$. Every predicate $\varphi \in \Phi$ partitions the variable valuations ν over Var into two parts: those that satisfy the predicate ($\nu \models \varphi$) and those that do not ($\nu \not\models \varphi$). Hence, n predicates in Φ split the (potentially infinitely many) variable valuations into finitely (up to 2^n) many blocks that are the equivalence classes of $\equiv_\Phi$ defined by

$$\nu \equiv_\Phi \nu' \quad \iff \quad (\nu \models \varphi \iff \nu' \models \varphi) \text{ for all } \varphi \in \Phi$$

for two variable valuations $\nu, \nu' \in \mathit{Val}(\mathit{Var})$. We lift this equivalence to the state space S by

$$\langle \ell, \nu \rangle \equiv_\Phi \langle \ell', \nu' \rangle \quad \iff \quad \ell = \ell' \wedge \nu \equiv_\Phi \nu'.$$

and let this define the partition Π. In other words, we treat states that disagree on their location component as *not* equivalent, because a different location typically indicates a significantly different behavior. However, this is merely a choice and may as well be resolved differently.

Recall that we assume the target states to be characterized by an expression η_T (see Section 6.2) and that for the game-based abstractions we require the target set T to be exactly representable. This can be easily achieved by including the predicate η_T in the predicate set Φ.

For a block $B \in \Pi$, we let $\ell(B)$ denote the (unique) location of the states in B. For a set of predicates $\Phi = \{\varphi_0, \dots, \varphi_{n-1}\}$, an equivalence class of $\equiv_\Phi$ can be represented using a bit vector of length n. That is, using the variables $\mathfrak{B} = \{b_i \mid 0 \leq i < n\}$ a block corresponds to a valuation $\nu \in \mathit{Val}(\mathfrak{B})$ together with the location $\ell(B)$ and we therefore also write a block B as a tuple $\langle \ell, \nu \rangle$ consisting of a location ℓ and a valuation $\nu \in \mathit{Val}(\mathfrak{B})$. In particular, we use the notation $B(b_i) \in \{0, 1\}$ to refer to the truth value of b_i in the block B.

Within the context of an abstraction-refinement loop, representing partitions using predicates also has the advantage that refinement is a matter of adding a suitable predicate to Φ. Note that with our assumption that all expressions in $\mathfrak{A}$ as well as the predicates are in LIA, an Smt solver can be used to determine whether two valuations are equivalent with respect to $\equiv_\Phi$.

For the remainder of this chapter, we assume a set of predicates $\Phi = \{\varphi_0, \dots, \varphi_{n-1}\}$ that represents the partition Π of the state space S induced by $\equiv_\Phi$. Furthermore, we assume copies $\mathit{Var}_1, \mathit{Var}_2, \dots$ of the variables in Var with $\mathit{Var}_i = \{x^i \mid x \in \mathit{Var}\}$ and let $\mathit{Var}^{\leq k} = \bigcup_{i=0}^{k} \mathit{Var}_i$ with $\mathit{Var}_0 = \mathit{Var}$. Similarly, we use the copies $\mathfrak{B}_1, \mathfrak{B}_2, \dots$ of the variables in $\mathfrak{B}$ with $\mathfrak{B}_j = \left\{b_i^j \mid b_i \in \mathfrak{B}\right\}$. Here, we also let $\mathfrak{B}_0 = \left\{b_i^0 \mid 0 \leq i < n\right\}$ with $b_i^0 = b_i$, so we have $\mathfrak{B}_0 = \mathfrak{B}$. Finally, we let $\mathfrak{B}^{\leq k} = \bigcup_{i=0}^{k} \mathfrak{B}_i$.

Example 54. Recall the partition Π from Example 50:

$$\Pi = \left\{ \underbrace{\{s_0\}}_{B_0}, \underbrace{\{s_1, s_3, s_5\}}_{B_1}, \underbrace{\{s_2, s_4\}}_{B_2}, \underbrace{\{s_6\}}_{B_3} \right\}.$$

This partition can be expressed using the predicates

$$\Phi = \left\{ \underbrace{phase = 0}_{\varphi_0}, \underbrace{phase = 1}_{\varphi_1}, \underbrace{phase = 2}_{\varphi_2}, \underbrace{phase = 3}_{\varphi_3} \right\}$$

that distinguish all values of the domain of *phase*. For example, B_1 consists of all the states $\langle \ell_0, v\rangle$ for which $v \vDash phase = 1$.

We remark that the partition induced by Φ is not strictly the same as Π, because Π is already restricted to the set of reachable states. As our goal is not to compute the set of reachable states but derive the abstraction directly from $\mathfrak{A}$, in practice we do not know which states are reachable. From now on, we assume Π to be induced by $\equiv_\Phi$ and therefore to range over *all* states. In our case, this amounts to $B_i = \left\{\langle \ell_0, v\rangle \in S^\# \mid v \vDash phase = i\right\}$.

6.4.2 Building Menu-Games from SPA

6.4.2.1 Logical Characterization of Abstract Semantics

Before we show how to phrase the derivation of menu-based abstraction in terms of Smt queries, we illustrate the same process for the concrete semantics. That is, we derive formulas from the edges of the SPA $\mathfrak{A}$ such that the solutions to these formulas correspond exactly to the choices in PA $\mathcal{M}$.

Definition 44 (Semantics of an Edge)**.** Let $e = \langle \ell, g, \alpha, D\rangle \in E$ be an edge. The semantics of e is given by

$$[\![e]\!] = \{\langle s, \alpha, \mu\rangle \mid s = \langle \ell, v\rangle \in S \wedge \mu = [\![D]\!]_{\mathfrak{A}}(v)\}.$$

Using the definition of the semantics of a (probabilistic) edge, the transition relation Δ of the PA $\mathcal{M}$ can be written as

$$\Delta = \bigcup_{e \in E} [\![e]\!].$$

For the rest of the chapter, we order the elements of the (finite) support of a symbolic probability distribution D as $supp(D) = \left\{\left\langle a_1^D, \ell_1^D \right\rangle, \ldots, \left\langle a_h^D, \ell_h^D \right\rangle\right\}$ with $h = |D|$.

Definition 45 (Transition Constraint)**.** Let $e = \langle \ell, g, \alpha, D \rangle \in E$ be a (probabilistic) edge. The *transition constraint* for e is

$$\varphi_e = g \wedge \bigwedge_{i=1}^{|D|} Var_i = a_i^D$$

where $Var_i = a_i^D$ is a shorthand for

$$\bigwedge_{\substack{x \in Var, \\ a_i^D(x) \neq \bot}} x^i = a_i^D(x) \quad \wedge \quad \bigwedge_{\substack{x \in Var, \\ a_i^D(x) = \bot}} x^i = x.$$

Lemma 4. *Let $e = \langle \ell, g, \alpha, D \rangle \in E$ be an edge. Then, for $v \in Val(Var^{\leq |D|})$*

$$v \vDash \varphi_e \iff \langle \langle \ell, v|_{Var} \rangle, \alpha, \mu_v \rangle \in [\![e]\!]$$

where

$$\mu_v(\ell', v') = \sum_{i=1}^{|D|} \begin{cases} D(a_i^D, \ell_i^D) & \textit{if } \ell' = \ell_i^D \textit{ and for all } x \in Var \,.\, v'(x) = v(x^i) \\ 0 & \textit{otherwise.} \end{cases}$$

Consequently, φ_e logically characterizes the semantics of e:

$$[\![e]\!] = \{ \langle \langle \ell, v|_{Var} \rangle, \alpha, \mu_v \rangle \mid v \vDash \varphi_e \}.$$

Example 55. Reconsider the SPA from Example 49. In particular, consider the edge $e_\beta = \left\langle \ell_0, phase = 1 \wedge run > 0, \beta, D_\beta \right\rangle$ with

$$D_\beta(a, \ell_0) = \begin{cases} 97/100 & \text{if } a = a_1 \\ 3/100 & \text{if } a = a_2 \\ 0 & \text{otherwise} \end{cases}$$

and $a_1 = a_\perp[run \mapsto run - 1]$, $a_2 = a_\perp[phase \mapsto 3]$. A corresponding transition constraint must refer to three variable sets $Var_0 = Var = \{phase, run\}$, $Var_1 = \{phase^1, run^1\}$ and $Var_2 = \{phase^2, run^2\}$, where Var_0 are the "source-state variables instances" and the others are the "destination-state variables instances" – one instance per assignment.

The transition constraint φ_{e_β} is given as

$$\underbrace{phase = 1 \wedge run > 0}_{g(e_\beta)} \wedge \underbrace{phase^1 = phase \wedge run^1 = run - 1}_{Var_1 = a_1} \wedge \underbrace{phase^2 = 3 \wedge run^2 = run}_{Var_2 = a_2} .$$

This constraint symbolically characterises the semantics of e_β, since its solutions induce $[\![e_\beta]\!]$. For example, the solution v with

$$\begin{aligned} v(phase) &= 1 & v(run) &= 2 \\ v(phase^1) &= 1 & v(run^1) &= 1 \\ v(phase^2) &= 3 & v(run^2) &= 2 \end{aligned}$$

corresponds to the tuple $\langle s_1, \beta, \mu_1 \rangle \in [\![e_\beta]\!]$ (see Figure 6.2).

Since φ_e characterizes the transitions of $\mathcal{M}$ that are generated by the edge e, one could, in principle, enumerate all solutions to φ_e for all edges e and obtain all choices of $\mathcal{M}$.

As we would like to enumerate the solutions of a formula to obtain the *abstract* behavior of the edge e, we now turn to define the abstract semantics of an edge.

Definition 46 (Abstract Semantics of an Edge)**.** Let $e = \langle \ell, g, \alpha, D \rangle \in E$ be an edge. The abstract semantics of e is given by

$$[\![e]\!]^\# = \{\langle B, \alpha, \mu' \rangle \mid B \in \Pi \wedge \exists \langle \ell, v \rangle \in B \,.\, \mu = [\![D]\!]_{\mathfrak{A}}(v) \wedge \mu' = [\mu]_\Pi \} .$$

Before formulating a formula that captures the abstract semantics of an edge, we formulate an essential building block. Recall that the partition Π is represented using a set $\Phi = \{\varphi_0, \ldots, \varphi_{n-1}\}$ of n predicates and that one block of the partition corresponds to a bit vector over the variables $\mathfrak{B}$ (together with the unique location of B). We let

$$\mathcal{B}_j = \bigwedge_{i=0}^{n-1} \left(b_i^j \iff \varphi_i \right)$$

and remark that the solutions of $\mathcal{B}_0$ characterize the non-empty blocks of the partition Π in terms of the variables $\mathfrak{B}_0 = \mathfrak{B}$.

Let $S^\# = Loc \times Val(\mathfrak{B})$ denote the abstract state space. For a given Boolean expression $\eta \in Bxp(Var)$, we can logically characterize the set of blocks containing a state $\langle \ell, v \rangle \in S$ with $v \models \eta$ by

$$[\![\eta]\!]^\# = \left\{ \langle \ell, v \rangle \in S^\# \mid v \models \varphi_\eta^\# \right\},$$

where $\varphi_\eta^\# = \eta \wedge \mathcal{B}_0$ is the *abstract expression constraint* for η. For convenience, we furthermore let $[\![\eta]\!]^\#(\ell)$ denote all blocks in $[\![\eta]\!]^\#$ that have location ℓ.

Similar to the transition constraint that expresses the semantics of an edge, we now define the *abstract transition constraint* that captures the abstract semantics of an edge.

Definition 47 (Abstract Transition Constraint)**.** Let $e = \langle \ell, g, \alpha, D \rangle \in E$ be an edge. The *abstract transition constraint* for e is

$$\varphi_e^\# = g \wedge \mathcal{B}_0 \wedge \bigwedge_{j=1}^{|D|} wp(\mathcal{B}_j, a_j^D).$$

Here, we use the weakest precondition of $\mathcal{B}_j$ with respect to the jth assignment $a_j^D(x)$ of the distribution D. This replaces all occurrences of variables of Var in $\mathcal{B}_j$ by their assigned expressions according to $a_j^D(x)$. Intuitively, this encodes the behavior of the assignments with respect to the target predicates in terms of the source state variables. Using this definition, the following holds.

Lemma 5. *Let $e = \langle \ell, g, \alpha, D \rangle \in E$ be an edge. Then, for $v \in Val(\mathfrak{B}^{\leq |D|})$*

$$v \models \varphi_e^\# \iff \left\langle \langle \ell, v|_{\mathfrak{B}} \rangle, \alpha, \mu_v^\# \right\rangle \in [\![e]\!]^\#$$

where

$$\mu_v^\#(\ell', v') = \sum_{j=1}^{|D|} \begin{cases} D(a_j^D, \ell_j^D) & \text{if } \ell' = \ell_j^D \text{ and for all } 0 \leq i < n \,.\, v'(b_i) = v(b_i^j) \\ 0 & \text{otherwise.} \end{cases}$$

Consequently, $\varphi^\#_e$ logically characterizes the abstract semantics of e:

$$[\![e]\!]^\# = \left\{ \left\langle \langle \ell, \nu|_{\mathfrak{B}} \rangle, \alpha, \mu^\#_\nu \right\rangle \mid \nu \models \varphi^\#_e \right\}.$$

Example 56. Reconsider the edge e_β and its symbolic probability distribution D_β from Example 55 and the set or predicates Φ from Example 54. The abstract transition constraint $\varphi^\#_{e_\beta}$ is

$$\begin{aligned}
&phase = 1 \wedge run > 0 && \big\}\ g(e_\beta)\\
&\left.\begin{aligned}&\wedge \left(b_0^0 \iff phase = 0\right) \wedge \left(b_0^1 \iff phase = 1\right)\\ &\wedge \left(b_0^2 \iff phase = 2\right) \wedge \left(b_0^3 \iff phase = 3\right)\end{aligned}\right\} && \mathcal{B}_0\\
&\left.\begin{aligned}&\wedge \left(b_1^0 \iff phase = 0\right) \wedge \left(b_1^1 \iff phase = 1\right)\\ &\wedge \left(b_1^2 \iff phase = 2\right) \wedge \left(b_1^3 \iff phase = 3\right)\end{aligned}\right\} && wp(\mathcal{B}_1, a_1)\\
&\left.\begin{aligned}&\wedge \left(b_2^0 \iff 3 = 0\right) \wedge \left(b_2^1 \iff 3 = 1\right)\\ &\wedge \left(b_2^2 \iff 3 = 2\right) \wedge \left(b_2^3 \iff 3 = 3\right)\end{aligned}\right\} && wp(\mathcal{B}_2, a_2)
\end{aligned}$$

We observe that B_0 and $wp(\mathcal{B}_1, a_1)$ differ only in the variables used to encode the target block. This is because the assignment a_1 of D_β (Example 55) only changes the value of the variable *run*, but this change cannot be observed using the predicates Φ that only refer to *phase*.

Now, it is easy to see that there is exactly one solution ν of this constraint over the variables $\mathfrak{B}^{\leq 2}$, namely

$$\begin{aligned}
&\nu(b_0^0) = \nu(b_0^2) = \nu(b_0^3) = \nu(b_1^0) = \nu(b_1^2) = \nu(b_1^3) = \nu(b_2^0) = \nu(b_2^1) = \nu(b_2^2) = 0\\
&\nu(b_0^1) = \nu(b_1^1) = \nu(b_2^3) = 1.
\end{aligned}$$

The projection of ν to the variables in $\mathfrak{B}$ corresponds to the source predicate valuation. As only $\nu(b_0^1) = 1$, i. e. $phase = 1$, the source block is B_1. By looking at the variables $\mathfrak{B}_1$ we can read off that the first assignment leads back to B_1. Similarly, we determine the successor block of the second assignment to be B_2, i. e. the block for which $phase = 3$

is true. Consequently, we have

$$\mu_v^\#(\ell_0, v') = \begin{cases} {}^{97}\!/_{100} & \text{if } v'(\varphi_i) = 1 \iff i = 1 \\ {}^{3}\!/_{100} & \text{if } v'(\varphi_i) = 1 \iff i = 3 \\ 0 & \text{otherwise} \end{cases}$$

and therefore $\mu_v^\#(\ell_0, v') = [\mu_1]_\Pi$ available in B_1 (see Figure 6.4).

6.4.2.2 Logical Characterization of Menu-Games

We just showed that we can derive the abstract (or concrete) semantics of edges by enumerating the solutions of logical constraints $\varphi_e^\#$ (or φ_e). It remains to illustrate that we can phrase the definition of the menu-game $\mathcal{G}^{\text{mba}}_{[\![\mathfrak{A}]\!],\Pi}$ in terms of our logical characterization of edges in the SPA $\mathfrak{A}$. The following lemma [Wac11] bridges this gap.

Lemma 6. *The menu-game*

$$\mathcal{G}^{\text{mba}}_{[\![\mathfrak{A}]\!],\Pi} = \left\langle \mathcal{V} = \mathcal{V}_1 \uplus \mathcal{V}_2 \uplus \mathcal{V}_p, \mathcal{E}, \mathcal{V}_0 \right\rangle$$

can be obtained with

» $\mathcal{V}_1 = \Pi \uplus \left\{v_\perp^1\right\}$,

» $\mathcal{V}_2 = \bigcup_{e \in E} \{\langle v_1, \alpha\rangle \in \mathcal{V}_1 \times Act \mid \langle v_1, \alpha, \mu\rangle \in [\![e]\!]^\#\} \uplus \left\{v_\perp^2\right\}$,

» $\mathcal{V}_p = \bigcup_{e \in E} \{\mu \mid \langle v_1, \alpha, \mu\rangle \in [\![e]\!]^\#\}$,

» $\mathcal{V}_0 = [\![Init^0 \wedge \bigwedge_{x \in Var} x \in Dom(x)]\!]^\#(\ell^0)$,

and the edge relation

$$\begin{aligned} \mathcal{E} = & \bigcup_{e \in E} \left\{\langle v_1, v_2\rangle \mid \ell = src(e) \wedge v_1 \in [\![g(e)]\!]^\#(\ell) \wedge v_2 = \langle v_1, \alpha(e)\rangle\right\} \\ \cup & \bigcup_{e \in E} \left\{\left\langle v_2, v_p\right\rangle \mid v_2 = \langle v_1, \alpha\rangle \wedge \left\langle v_1, \alpha(e), v_p\right\rangle \in [\![e]\!]^\#\right\} \\ \cup & \bigcup_{e \in E} \left\{\left\langle v_2, v_\perp^p\right\rangle \mid v_2 = \langle v_1, \alpha(e)\rangle \wedge v_1 \in [\![\neg g(e)]\!]^\#\right\} \\ \cup & \left\{\left\langle v_\perp^1, v_\perp^2\right\rangle, \left\langle v_\perp^2, v_\perp^p\right\rangle\right\} \\ \cup & \left\{\left\langle v_p, v'\right\rangle \mid v_p(v') > 0\right\}. \end{aligned}$$

In other words, the edge relation $\mathcal{E}$ can be expressed solely in terms of $[\![b]\!]^{\#}$ for $b \in Bxp(Var)$ and $[\![e]\!]^{\#}$ for an edge $e \in E$.

6.4.2.3 Algorithmically Building Menu-Games

These formulae can be dispatched to an off-the-shelf SMT solver that supports the required theory. More concretely, we can use ALLSAT procedures

» over the variables $\mathfrak{B}^0$ to enumerate the solutions of $\varphi^{\#}_{\eta}$ to determine $[\![\eta]\!]^{\#}$ for expressions $\eta \in Bxp(Var)$, and

» over the variables $\mathfrak{B}^{\leq k}$ to enumerate the solutions of $\varphi^{\#}_{e}$ to determine $[\![e]\!]^{\#}$ for an edge e whose symbolic probability distribution has a support of size k.

Since edges can be considered in isolation, we first formulate building an MTBDD characterizing the choices in $\mathcal{G}^{\mathrm{mba}}_{\mathcal{M},\Pi}$ that are associated with a single edge $e \in E$. Algorithm 8 uses the aforementioned ALLSAT enumeration in lines 3 and 18 to realize this. Lines 3 to 7 build a cache that maps source (player 1) vertices of the game to their choices in terms of player 2, probabilistic and player 1 successor vertices. The next block (lines 9 to 14) combines the entries of the cache to an MTBDD M_e that represents all choices associated with edge e without the transitions to $v^1_{\perp}$. These transitions are computed in the last block (lines 17 to 20) before the overall result is constructed in line 21.

In a second step, we develop Algorithm 9 that combines the MTBDDs representing the menu-game fragments related to single edges to the MTBDD $\mathsf{M}^{\#}_{\mathfrak{A}}$ that represents the full menu-game. It uses Algorithm 8 as a subroutine and combines the subresults in lines 3 and 4 before adding the "loop" at $v^1_{\perp}$ (via $v^2_{\perp}$ and $v^p_{\perp}$) in line 5.

Algorithm 9 can be easily extended to perform a reachability analysis and remove all unreachable vertices from the game. As this is a standard procedure, we do not go into details here and simply assume that the resulting game is restricted to its reachable fragment.

Mostly for completeness reasons, we include Algorithm 10 that takes a Boolean expression η over the variables *Var* and computes the BDD representation B_{η} of $[\![\eta]\!]^{\#}$. Depending on the expression η, it serves several purposes. For example, choosing $\eta = Init^0 \wedge \bigwedge_{x \in Var} x \in Dom(x)$ yields a BDD that characterizes the initial vertices of the abstract game. Similarly, it can be used to derive the abstract target vertices, because the

Algorithm 8: Abstraction of a single edge.

```
function AbstractEdge($e = \langle \ell, g, \alpha, D \rangle$):
    input: $e$: a (probabilistic) edge of the SPA $\mathfrak{A}$ to abstract
    output: an MTBDD representing the abstraction of the edge
    $cache \leftarrow \varnothing$
    foreach $\langle v_1, \alpha, \mu \rangle \in [\![e]\!]^{\#}$ do                // enumerate abstract semantics of edge
        $\mathsf{M}_\mu \leftarrow \textsc{Const}(0)$
        foreach $v' \in supp(\mu)$ do
            $\mathsf{M}_\mu \leftarrow \mathsf{M}_\mu + \textsc{Encode}(\mathcal{S}' \leftarrow \langle v' \rangle) \cdot \textsc{Const}(\mu(v'))$
        $cache[v_1] \leftarrow cache[v_1] \cup \{\mathsf{M}_\mu\}$

    $\mathsf{M}_e \leftarrow \textsc{Const}(0)$
    foreach $\langle v_1, \{\mathsf{M}_\mu^1, \dots, \mathsf{M}_\mu^m\} \rangle \in cache$ do                // combine distributions to $\mathsf{M}_e$
        $\mathsf{M} \leftarrow \textsc{Const}(0)$
        for $i \in \{1, \dots, m\}$ do
            $\mathsf{M} \leftarrow \mathsf{M} + \mathsf{M}_\mu^i \cdot \textsc{Encode}(\mathcal{O} \leftarrow \langle i \rangle)$
        $\mathsf{M}_e \leftarrow \mathsf{M}_e + \textsc{Encode}(\mathcal{S} \leftarrow \langle v_1 \rangle) \cdot \mathsf{M}$
    $\mathsf{B}_e \leftarrow \textsc{ExistAbstract}(\mathcal{A} \uplus \mathcal{O} \uplus \mathcal{S}', \mathsf{M}_e \neq 0)$                // compute source blocks

    $\mathsf{M}_\perp \leftarrow \textsc{Const}(0)$
    foreach $v_1 \in [\![\neg g]\!]^{\#}$ do                // determine transitions to $v_\perp^1$
        $\mathsf{M}_\perp \leftarrow \mathsf{M}_\perp + \textsc{Encode}(\mathcal{S} \leftarrow \langle v_1 \rangle, \mathcal{O} \leftarrow \langle \langle v_1, \langle v_1, \alpha \rangle, \delta_{v_\perp^1} \rangle \rangle, \mathcal{S}' \leftarrow \langle v_\perp^1 \rangle)$
    $\mathsf{M}_\perp \leftarrow \mathsf{B}_e \cdot \mathsf{M}_\perp$
    return $(\mathsf{M}_e + \mathsf{M}_\perp) \cdot \textsc{Encode}(\mathcal{A} \leftarrow \langle \alpha \rangle)$
```

Algorithm 9: Abstraction of an SPA.

```
1 function AbstractModel($\mathfrak{A}$):
      input: $\mathfrak{A}$: the SPA to abstract with edge set $E$
      output: an MTBDD representing the abstraction of the SPA $\mathfrak{A}$
2     $\mathsf{M}^{\#}_{\mathfrak{A}} \leftarrow$ Const $(0)$
3     foreach $e \in E$ do                    // combine MTBDDs for abstract behavior
4         $\mathsf{M}^{\#}_{\mathfrak{A}} \leftarrow \mathsf{M}^{\#}_{\mathfrak{A}} +$ AbstractEdge$(e)$
5     $\mathsf{M}^{\#}_{\mathfrak{A}} \leftarrow \mathsf{M}^{\#}_{\mathfrak{A}} \cdot$ Encode $\left(\mathcal{S} \leftarrow \langle v^1_\perp \rangle, \mathcal{A} \leftarrow \langle v^2_\perp \rangle, \mathcal{O} \leftarrow \langle v^p_\perp \rangle, \mathcal{S}' \leftarrow \langle v^1_\perp \rangle\right)$
6     return $\mathsf{M}^{\#}_{\mathfrak{A}}$
```

Algorithm 10: Abstraction of an expression.

```
1 function AbstractExpression($\eta$):
      input: $\eta$: an expression from $Bxp(Var)$ to abstract
      output: a BDD representing the abstraction $[\![\eta]\!]^{\#}$
2     $\mathsf{B}_\eta \leftarrow$ Const $(0)$
3     foreach $v_1 \in [\![\eta]\!]^{\#}$ do                    // enumerate abstract blocks
4         $\mathsf{B}_\eta \leftarrow \mathsf{B}_\eta +$ Encode $(\mathcal{S} \leftarrow \langle v_1 \rangle)$
5     return $\mathsf{B}_\eta$
```

target set T is given in terms of an expression η_T over the variables of $\mathfrak{A}$ (and a target location ℓ_T).

6.4.3 Solving Menu-Games

Given the menu-based abstraction $\mathcal{G} = \mathcal{G}^{\text{mba}}_{\mathcal{M},\Pi}$ of $\mathcal{M}$, Theorem 3 states that the minimal and maximal reachability probabilities in $\mathcal{M}$ with respect to the target state T are bounded from below and above by reachability probabilities in $\mathcal{G}$. In general, the first player minimizes or maximizes based on the direction of the property and the second player minimizes or maximizes based on whether the lower or upper bound is to be computed. In the minimizing case, the abstract target set $T^{\#}$ is extended with $v^1_\perp$ to account for the peculiarities of the menu-based subtransformer $me^{-}_{\mathcal{M},T}[\alpha]$.

Given the MTBDD $\mathsf{M}^{\#}_{\mathfrak{A}}$ that represents the menu-based abstraction $\mathcal{G}$, the next step is to compute the corresponding reachability probabilities with respect to the target vertices. We refer to the abstract target set by $T^{\#}$ and pretend that it includes $v^1_\perp$ in case of bounding minimal reachability probabilities in $\mathcal{M}$. As we use the algorithms presented in this section in the particular context of menu-based abstraction of PA, we choose to maintain the same notation as before, but remark that the algorithms are applicable to general (finite) SGs.

6.4.3.1 Qualitative Solution

Similar to standard approaches to model checking PA, we break the solution process up into two parts. In the first step, we solve the game *qualitatively*. That is, we compute all (player 1) vertices with reachability probability 0 or 1 given the optimization directions $\star_1$ and $\star_2$ of the two players. Just like for PA, the algorithms for this abstract from the actual probabilities and only need to consider the graph structure (i. e. the edge relation $\mathcal{E}$) of the game.

We start with Algorithm 11 (PROB0) that determines the vertices that have probability 0 to reach the target states $T^\#$. The approach of the algorithm is to perform a backward search through the game starting from vertices in $T^\#$ and add all vertices encountered this way to the set $\mathcal{V}_1^{>0}$. Ultimately, this set contains all vertices, which have a probability strictly *greater* than 0 and the result can therefore be obtained by inverting $\mathcal{V}_1^{>0}$ with respect to $\mathcal{V}_1$.

The algorithm can be understood as a fixed-point iteration that proceeds as long as new vertices were found with probability greater than 0. In this backward iteration, it depends on the optimization directions of the players which vertices are added. Roughly speaking, if a player maximizes ($\star = +$) it suffices that there *is* at least one successor whose reachability probability is known to be greater than 0. In contrast, when minimizing ($\star = -$), a vertex is only added if *all* successors have probability greater than 0. Intuitively, this is because the minimizing player tries to avoid such successor vertices whenever possible.

We now turn to Algorithm 12 (PROB1) that computes all (player 1) vertices from which the probability to reach $T^\#$ with the two optimization directions $\star_1$ and $\star_2$ is 1. In contrast to PROB0, PROB1 requires a double fixed-point computation, which is not surprising as this is already the case for PA. Intuitively, the reason for the double fixed-point is that there may be mutual dependencies between vertices that govern whether these vertices are in the solution or not. In particular, it is too restrictive to amend the approach of Algorithm 11 by replacing the (implicit) existential quantification in line 6 with a universal quantification (and not inverting the result in line 16). This would, for example, not find vertices that probabilistically branch to a target state and itself even though these vertices clearly have a reachability probability equal to 1.

Algorithm 11: Compute vertices with probability 0.

```
function Prob0($\mathcal{G} = \langle \mathcal{V} = \mathcal{V}_1 \uplus \mathcal{V}_2 \uplus \mathcal{V}_p, \mathcal{E}, \mathcal{V}_0 \rangle, \star_1, \star_2, T^\# \subseteq \mathcal{V}_1$):
    input: $\mathcal{G}$: the SG for which to compute the vertices with probability 0,
           $\star_1$: the optimization direction of player 1,
           $\star_2$: the optimization direction of player 2,
           $T^\#$: the set of abstract target vertices
    output: the (player 1) vertices with probability 0
    $\mathcal{V}_1^{>0,old} \leftarrow \emptyset$
    $\mathcal{V}_1^{>0} \leftarrow T^\#$
    while $\mathcal{V}_1^{>0} \neq \mathcal{V}_1^{>0,old}$ do        // fixpoint iteration until no new vertices found
        $\mathcal{V}_1^{>0,old} \leftarrow \mathcal{V}_1^{>0}$
        $\mathcal{V}_p^{>0} \leftarrow \{v_p \in \mathcal{V}_p \mid \mathcal{V}_1^{>0} \cap succ_{\mathcal{G}}(v_p) \neq \emptyset\}$        // perform probabilistic step
        if $\star_2 = +$ then        // perform player 2 step
            $\mathcal{V}_2^{>0} \leftarrow \{v_2 \in \mathcal{V}_2 \mid succ_{\mathcal{G}}(v_2) \cap \mathcal{V}_p^{>0} \neq \emptyset\}$
        else
            $\mathcal{V}_2^{>0} \leftarrow \{v_2 \in \mathcal{V}_2 \mid succ_{\mathcal{G}}(v_2) \subseteq \mathcal{V}_p^{>0}\}$
        if $\star_1 = +$ then        // perform player 1 step
            $\mathcal{V}_1^{>0} \leftarrow \{v_1 \in \mathcal{V}_1 \mid succ_{\mathcal{G}}(v_1) \cap \mathcal{V}_2^{>0} \neq \emptyset\}$
        else
            $\mathcal{V}_1^{>0} \leftarrow \{v_1 \in \mathcal{V}_1 \mid succ_{\mathcal{G}}(v_1) \subseteq \mathcal{V}_2^{>0}\}$
        $\mathcal{V}_1^{>0} \leftarrow \mathcal{V}_1^{>0} \cup T^\#$        // re-add target vertices
    return $\mathcal{V}_1 \setminus \mathcal{V}_1^{>0}$        // invert solution w.r.t. player 1 vertices
```

Algorithm 12: Compute vertices with probability 1.

```
1  function Prob1($\mathcal{G} = \langle \mathcal{V} = \mathcal{V}_1 \uplus \mathcal{V}_2 \uplus \mathcal{V}_p, \mathcal{E}, \mathcal{V}_0 \rangle, \star_1, \star_2, T^\# \subseteq \mathcal{V}_1$):
       input: $\mathcal{G}$: the SG for which to compute the vertices with probability 1,
              $\star_1$: the optimization direction of player 1,
              $\star_2$: the optimization direction of player 2,
              $T^\#$: the set of abstract target vertices
       output: the (player 1) vertices with probability 1
2      $\mathcal{V}_1^{maybe} \leftarrow \mathcal{V}_1$
3      $\mathcal{V}_1^{=1} \leftarrow T^\#$
4      while $\mathcal{V}_1^{maybe} \neq \mathcal{V}_1^{=1}$ do                    // outer fixpoint iteration
5          $\mathcal{V}_1^{maybe} \leftarrow \mathcal{V}_1^{=1}$
6          $\mathcal{V}_1^{=1} \leftarrow T^\#$
7          $\mathcal{V}_1^{=1,old} \leftarrow \emptyset$
8          while $\mathcal{V}_1^{=1} \neq \mathcal{V}_1^{=1,old}$ do                // inner fixpoint iteration
9              $\mathcal{V}_1^{=1,old} \leftarrow \mathcal{V}_1^{=1}$
               // perform probabilistic step
10             $\mathcal{V}_p^{=1} \leftarrow \left\{ v_p \in \mathcal{V}_p \mid succ_{\mathcal{G}}(v_p) \subseteq \mathcal{V}_1^{maybe} \wedge succ_{\mathcal{G}}(v_p) \cap \mathcal{V}_1^{=1} \neq \emptyset \right\}$
11             if $\star_2 = +$ then                                // perform player 2 step
12                 $\mathcal{V}_2^{=1} \leftarrow \left\{ v_2 \in \mathcal{V}_2 \mid succ_{\mathcal{G}}(v_2) \cap \mathcal{V}_p^{=1} \neq \emptyset \right\}$
13             else
14                 $\mathcal{V}_2^{=1} \leftarrow \left\{ v_2 \in \mathcal{V}_2 \mid succ_{\mathcal{G}}(v_2) \subseteq \mathcal{V}_p^{=1} \right\}$
15             if $\star_1 = +$ then                                // perform player 1 step
16                 $\mathcal{V}_1^{=1} \leftarrow \left\{ v_1 \in \mathcal{V}_1 \mid succ_{\mathcal{G}}(v_1) \cap \mathcal{V}_2^{=1} \neq \emptyset \right\}$
17             else
18                 $\mathcal{V}_1^{=1} \leftarrow \left\{ v_1 \in \mathcal{V}_1 \mid succ_{\mathcal{G}}(v_1) \subseteq \mathcal{V}_2^{=1} \right\}$
19             $\mathcal{V}_1^{=1} \leftarrow \mathcal{V}_1^{=1} \cup T^\#$                 // re-add target vertices
20     return $\mathcal{V}_1^{=1}$
```

Algorithm 13: Implementation of the value iteration transformers $vi_{\mathcal{G},T^\#}^{\star_1\star_2}$.

```
function ValueIterationStep($\mathcal{G} = \langle \mathcal{V} = \mathcal{V}_1 \uplus \mathcal{V}_2 \uplus \mathcal{V}_p, \mathcal{E}, \mathcal{V}_0 \rangle$, $\star_1$, $\star_2$, $T^\#$, $w^\#$):
    input: $\mathcal{G}$: the SG for which perform the value iteration step,
           $\star_1$: the optimization direction of player 1,
           $\star_2$: the optimization direction of player 2,
           $T^\#$: the set of abstract target vertices,
           $w^\#$: the initial abstract (player 1) vertex valuation
    output: the abstract valuation after one application of $vi_{\mathcal{G},T^\#}^{\star_1\star_2}$
    foreach $v_1 \in \mathcal{V}_1$ do
        if $v_1 \in T^\#$ then
            $w^\#(v_1) \leftarrow 1$                    // explicitly set values of target states
        else
            $w^\#(v_1) \leftarrow \bigstar_{1\, v_2 \in succ_{\mathcal{G}}(v_1)} \bigstar_{2\, v_p \in succ_{\mathcal{G}}(v_2)} \sum_{v_1' \in \mathcal{V}_1} v_p(v_1') \cdot w^\#(v_1')$
    return $w^\#$
```

The algorithm proceeds as follows. Starting with all player 1 vertices, the outer fixed point iteration shrinks a set of "maybe" vertices that are *potentially* in the solution set until a fixed point is reached that is then returned. The inner fixed point iteration performs a backward search from the target vertices. While for Prob0, the probabilistic "step" consisted of testing whether *there is* a successor in the current vertex set, for Prob1 it is additionally required that *all* transitions remain in the current "maybe" vertices. Other than that, the inner fixed-point iteration proceeds very similarly to that of Prob0 in that it resolves the players' choices in the same manner.

6.4.3.2 Quantitative Solution

It remains to solve vertices whose reachability probabilities lie strictly between 0 and 1. Even though there are other algorithms that solve this problem, we choose to stick to our previous presentations and approximate the desired probabilities using the value iteration transformers $vi_{\mathcal{G},T}^{\star_1\star_2}$ introduced in the beginning of this chapter. Algorithm 13 realizes these transformers in a straightforward manner.

Algorithm 14 embeds Algorithm 13 in a fixed-point algorithm that approaches the desired least fixed points from below. As pointed out earlier, the Kleene fixed-point theorem guarantees that starting from the constant zero vector, iterating the value iteration

Algorithm 14: Value iteration for stochastic games.

```
1 function ValueIteration($\mathcal{G} = \langle \mathcal{V} = \mathcal{V}_1 \uplus \mathcal{V}_2 \uplus \mathcal{V}_p, \mathcal{E}, \mathcal{V}_0 \rangle, \star_1, \star_2, \mathcal{V}_1^{=1}, T^\#, w^\#$):
    input: $\mathcal{G}$: the SG for which perform the value iteration step,
           $\star_1$: the optimization direction of player 1,
           $\star_2$: the optimization direction of player 2,
           $\mathcal{V}_1^{=1}$: the set of (player 1) vertices with probability 1,
           $T^\#$: the set of abstract target vertices,
           $w^\#$: the initial abstract (player 1) vertex valuation
    output: the abstract valuation approximating $\Pr_{\mathcal{G}}^{\star_1 \star_2}(T^\#)$
2   foreach $v_1 \in \mathcal{V}_1^{=1}$ do                    // set all precomputed values
3       $w^\#(v_1) \leftarrow 1$
4   repeat
5       $w^\#_{old} \leftarrow w^\#$
6       $w^\# \leftarrow$ ValueIterationStep($\mathcal{G}, \star_1, \star_2, T^\#, w^\#_{old}$)
7   until Converged($w^\#, w^\#_{old}, \mathcal{G}, \epsilon$)          // check for convergence
8   return $w^\#$
```

transformers converges to the least fixed-points of the transformers, which coincide with the corresponding reachability probabilities. This fixed-point approximation has to be aborted at some point and it turns out that it is not trivial to develop a stopping criterion that guarantees that the solution is within ϵ distance of the actual result. Most implementations choose to compare the values of the current and the last step and terminate if these two vectors are close enough to one another. This, however, does not give any guarantees regarding the preciseness of the current approximation with respect to the target value. This fundamental problem has recently drawn attention and been approached from different angles for PA [HM14; Bai+17b; QK18] as well as for SGs [Kel+18]. We remark that in order to obtain sound lower and upper bounds for the reachability probabilities in $\mathcal{M}$ using game-based or menu-based abstraction, actual implementations need to account for this problem or choose a different solution technique such as policy iteration [Con90]. However, we do not treat this issue further in our presentation here.

Finally, we connect Algorithms 11, 12 and 14 in a procedure that obtains both lower and upper bounds on the reachability probabilities in $\mathcal{M}$ shown in Algorithm 15. Since the upper bounds are known to be at least as large as the lower bounds, we start the value iteration for the upper bounds from the lower values rather than $0^{\mathcal{V}_1}$.

Algorithm 15: Computes lower and upper bounds for the $\star$-reachability in $\mathcal{M}$.

```
1  function ComputeBounds($\mathcal{G} = \mathcal{G}^{\mathrm{mba}}_{\mathcal{M},\Pi}$, $\star$, $T^{\#}$):
       input: $\mathcal{G}$: the menu-game $\mathcal{G}^{\mathrm{mba}}_{\mathcal{M},\Pi}$,
              $\star$: the optimization direction of the property,
              $T^{\#}$: the set of abstract target vertices
       output: a pair of (abstract) valuations that bound the reachability probabilities
       // Compute lower bounds
2      $\mathcal{V}^{l,=1}_1 \leftarrow$ Prob1($\mathcal{G}$, $\star$, $-$, $T^{\#}$)
3      $w^{\#}_l \leftarrow$ ValueIteration($\mathcal{G}$, $\star$, $-$, $\mathcal{V}^{l,=1}_1$, $T^{\#}$, $\mathbf{0}$)
4
       // Compute upper bounds
5      $\mathcal{V}^{u,=1}_1 \leftarrow$ Prob1($\mathcal{G}$, $\star$, $+$, $T^{\#}$)
6      $w^{\#}_u \leftarrow$ ValueIteration($\mathcal{G}$, $\star$, $+$, $\mathcal{V}^{u,=1}_1$, $T^{\#}$, $w^{\#}_l$)
7      return $\langle w^{\#}_l, w^{\#}_u \rangle$
```

6.4.4 Refinement

Recall the overall abstraction-refinement loop in Figure 6.5. Until now we have treated

(i) the fully automatic abstraction of the PA $\mathcal{M}$ directly from the SPA $\mathfrak{A}$ in terms of the menu-based abstraction $\mathcal{G}^{\mathrm{mba}}_{\mathcal{M},\Pi}$, and

(ii) the algorithmic solution of $\mathcal{G}^{\mathrm{mba}}_{\mathcal{M},\Pi}$ to obtain lower and upper bounds on both minimal and maximal reachability probabilities in $\mathcal{M}$ with respect to a set of target states T.

The result of this is one of two possible outcomes. First, the bounds may be precise enough to answer the current query. This could, for example be the case if the property specifies an upper bound λ on the reachability probability and the upper bound for the initial state obtained via the abstract game lies below λ. In this case we can infer that the bound is met in $\mathcal{M}$. In case the result is inconclusive, however, the abstraction needs to be refined.

As we argued earlier, predicate abstraction lends itself for obtaining the abstraction. Not only does it enable the use of Smt solvers to build the abstract system, but refining the system reduces to introducing suitable new predicates. We now show how to derive such predicates. For this, we assume that the distributions represented by the probabilistic vertices $\mathcal{V}_p$ are additionally annotated with the assignments from the SPA $\mathfrak{A}$ that led to the target (player 1) vertices. For this, we let $\widehat{\mathcal{V}_p} \subseteq \mathit{Dist}(\mathit{Asg}(\mathit{Var}) \times \mathcal{V}_1)$ be the extension of $\mathcal{V}_p$ with assignments. When building the abstraction using the approach in Section 6.4.2.3, it is is easy to additionally maintain the assignments responsible for the different successor vertices.

To realize the refinement, our presentation assumes that we have two strategy pairs $\langle \sigma_1^l, \sigma_2^l \rangle$ and $\langle \sigma_1^u, \sigma_2^u \rangle$ for $\mathcal{G} = \mathcal{G}^{\mathrm{mba}}_{\mathcal{M},\Pi}$ such that for $\star \in \{-,+\}$ we have

$$w_l^\# = \Pr_{\mathcal{G}}^{\star -}(\lozenge T^\#) = \Pr_{\mathcal{G}}^{\langle \sigma_1^l, \sigma_2^l \rangle}(\lozenge T^\#) \text{ and } \Pr_{\mathcal{G}}^{\star +}(\lozenge T^\#) = \Pr_{\mathcal{G}}^{\langle \sigma_1^u, \sigma_2^u \rangle}(\lozenge T^\#) = w_u^\#. \tag{6.6}$$

That is, resolution of the nondeterministic choices according to $\langle \sigma_1^l, \sigma_2^l \rangle$ results in the lower bounds $w_l^\#$ and resolution according to $\langle \sigma_1^u, \sigma_2^u \rangle$ yields the upper bounds $w_u^\#$. We refer to $\langle \sigma_1^l, \sigma_2^l \rangle$ and $\langle \sigma_1^u, \sigma_2^u \rangle$ as the lower and upper strategies, respectively. Here, we also assume that $w_l^\#$ and $w_u^\#$ are defined not only on $\mathcal{V}_1$ but also on $\mathcal{V}_2$ and $\mathcal{V}_p$ in the straightforward manner.

So, assuming such strategies are given, how are new predicates obtained? One tempting direction might be to look for vertices v_1 for which $w_l^\#(v_1) < w_u^\#(v_1)$. However, while the bounds may be imprecise, the vertex v_1 is not necessarily *the cause* of the imprecision. It may well be that the successor vertices of v_1 were the reason for the deviation of the lower and upper bounds at v_1. A better criterion therefore is to consider the strategies themselves. The strategies are tightly linked to the abstraction, because player 2 choices intuitively govern which states from the abstract block represented by v_1 is selected to realize the lower and upper bounds. We need to be a bit careful, though. In general, there may be several choices for the players whose achieved values agree. If the lower and upper strategies deviate, but ultimately represent the same value, this does not explain the deviation of the bounds at v_1. [Wac11] therefore requires that the strategies are constructed such that the upper strategies only differ from the lower strategies at vertices where this is "necessary". Formally, the authors require that for every $v_i \in \mathcal{V}_i$, $i \in \{1, 2\}$ the strategies satisfy

$$\sigma_i^l(v_i) \neq \sigma_i^u(v_i) \implies w_u^\#(\sigma_i^l(v_i)) < w_u^\#(v_i). \tag{6.7}$$

Intuitively, this criterion states that player i may only make different choices in v_i in the lower and upper strategies if the lower strategy successor $v = \sigma_i^l(v_i)$ achieves a strictly smaller value even when maximizing from v_i. In this case, the deviation of the strategies in v_i is therefore regarded as "justified".

We do not dwell on the details on how to obtain such strategies, but only briefly mention a few intricacies. First, we observe that during the qualitative solution of the abstract game (Algorithm 11 and Algorithm 12), it is not difficult to obtain strategies that realize the values 0 and 1 for the vertices that are the result of the respective algorithm. For Prob0, it is straightforward to select choices in vertices that stay within the set of states with probability 0. In the case of Prob1, it is not sufficient to locally pick actions that stay within the set of states with probability 1. Ultimately, the reason for this is the same as the one that necessitated the double fixed-point computation in Prob1. However, we can construct the strategy pair by requiring progress towards the target vertex set in the same way as the algorithm in the inner fixed-point iteration. Also, for $\mathcal{V}_1^{=1,\star_1,\star_2} = \{v_1 \in \mathcal{V}_1 \mid \Pr_{\mathcal{G}}^{\star_1\star_2}(\lozenge T^\#) = 1\}$ we have $\mathcal{V}_1^{=1,\star,-} \subseteq \mathcal{V}_1^{=1,\star,+}$, and we can therefore reuse the lower strategies for all $v_1 \in \mathcal{V}_1^{=1,\star,-}$.

Secondly, [Wac11] tries to extend the value iteration process such that the generated strategies not only realize the computed extremal probabilities (Equation (6.6)), but also satisfy Equation (6.7). This is done by paying special attention as to when the strategy of a vertex is updated. In a nutshell, the strategy of a vertex is only modified if the value strictly improved over successive iterations of the value iteration. This ensures that the choices do not stay within an end component of the system (relating to condition (6.6) above). If the latter were to happen, the selected choices would not realize the computed bounds, but rather induce zero reachability probability. Choosing such a strategy is clearly wrong in states with probability greater than 0 and is avoided by the more restrictive update process. However, in contrast to the claim, the extended algorithm [Wac11] does *not* establish condition (6.7). This issue arises from the fact that probability mass is "propagated at different speeds" through the system by value iteration. The value iteration by [Wac11] therefore may prematurely modify the strategy in a vertex even though ultimately the previous choice achieves the same probability.

We now show that it is not possible to *first* fix the lower strategies in the game to realize the lower bounds and *then* compute the upper bounds and strategies in a way that establishes condition (6.7), even when only enforcing it for player 2 vertices.

Example 57. Consider the SG $\mathcal{G}$ in Figure 6.6. We refrain from giving an SPA and suitable predicates to arrive at this game, but remark that it is possible to construct such entities. With $w_l^\# = \Pr_{\mathcal{G}}^{+-}(\lozenge T^\#)$, we have $w_l^\#(v_1^0) = w_l^\#(v_1^1) = 0$. If the solution process derives a lower strategy pair $\langle \sigma_1^l, \sigma_2^l \rangle$ that realizes this probability for v_1^0 and v_1^1, then $\sigma_2^l(v_2^2) = v_p^2$. However, in v_1^0, there is a choice for player 1 and we might have $\sigma_1^l(v_1^0) = v_2^0$ and $\sigma_1^l(v_1^1) = v_2^2$, i. e. player 1 chooses α in v_1^0 and γ in v_1^1. Similarly, for $w_u^\# = \Pr_{\mathcal{G}}^{++}(\lozenge T^\#)$ we have $w_u^\#(v_1) = w_u^\#(v_1^1) = 1$ and the upper strategy pair $\langle \sigma_1^u, \sigma_2^u \rangle$ needs to satisfy $\sigma_1^u(v_1^0) = v_2^1$. However, this is not possible under Equation (6.7). To

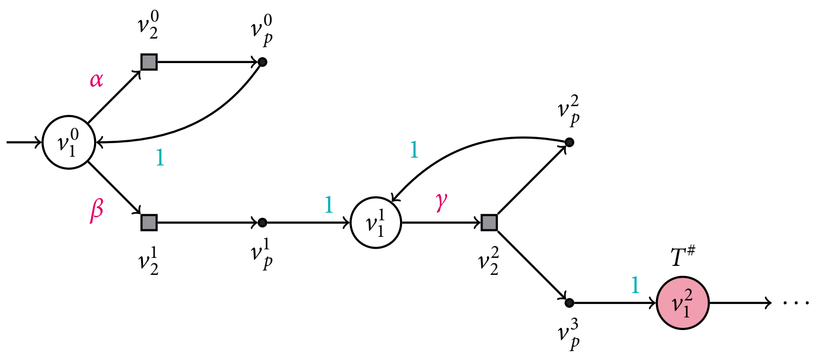


Figure 6.6: An example SG for which Equation (6.7) is problematic.

see this, we observe $v_2^0 = \sigma_1^l(v_1^0) \neq \sigma_1^u(v_1^0) = v_2^1$, so Equation (6.7) requires

$$w_u^\#(\sigma_1^l(v_1^0)) = w_u^\#(v_2^0) < w_u^\#(v_2^1) = w_u^\#(v_1^0).$$

However, this can not be satisfied as $w_u^\#(v_2^0) = w_u^\#(v_1^0) = w_u^\#(v_2^1)$.

We now go on to show that it is not trivial to establish property (6.7) in a post-processing step either. It might be tempting to fix the violation of Equation (6.7) in a player 2 vertex $v_2 \in \mathcal{V}_2$ by falsifying the condition's premise. More concretely, one could argue that $w_u^\#(\sigma_2^l(v_2)) = w_u^\#(v_2)$ is sufficient to infer that the upper choice of player 2 in v_2 may be set to the lower choice, because — intuitively speaking — the lower choice is sufficient to realize the same upper bound $w_u^\#(v_2)$. This is not true as the following example shows.

Example 58. Consider the SG $\mathcal{G}$ in Figure 6.7. Again, we refrain from giving an SPA and suitable predicates to arrive at this game, but remark that it is possible to construct such entities. We annotate all vertices v with the interval $[w_l^\#(v), w_u^\#(v)]$. First of all, we observe that player 1 does not have choices in this SG. Now, to realize the lower bound in v_1^0, the lower strategy of player 2 could pick $\sigma_2^l(v_2^0) = v_p^0$. Similarly, player 2 can choose to move from v_2^0 to v_p^1 to realize the upper bound $w_u^\#(v_2^0)$. Then, the choices of player 2 differ in the lower and upper strategy and Equation (6.7)

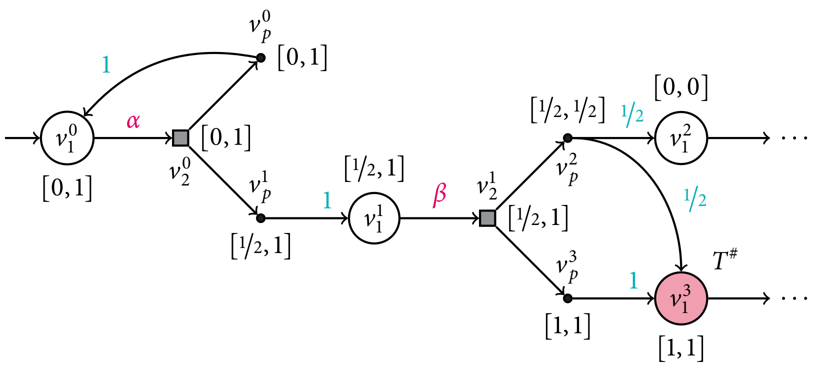


Figure 6.7: An example SG for which Equation (6.7) is problematic.

therefore requires

$$w_u^\#(\sigma_2^l(v_2^0)) = w_u^\#(v_p^0) = 1 < 1 = w_u^\#(v_2^0),$$

which is not satisfied.

As previously mentioned, it might be tempting to "repair" condition (6.7) by switching the upper to the lower choice in v_2^0 since the condition seemingly implies that the lower choice achieves an equally good upper bound. However, setting $\sigma_2^u(v_2^0) = v_p^0$ achieves a reachability probability of 0 and no longer realizes the upper bound $w_u^\#(v_2^0) = 1$.

In fact, for the given SG, *all* (player 2) strategies that realize the lower bound in v_2^0 need to move to v_p^0 and *all* (player 2) strategies that realize the upper bound in v_2^0 need to move to v_p^1. Hence, by the reasoning above, it is impossible to both realize the lower and upper bounds and simultaneously satisfy Equation (6.7).

Because of the aforementioned problems, we propose to consider the structurally very similar criterion

$$\sigma_2^l(v_2) \neq \sigma_2^u(v_2) \implies w_l^\#(\sigma_2^l(v_2)) < w_l^\#(\sigma_2^u(v_2)) \text{ for all } v_2 \in \mathcal{V}_2 \tag{6.8}$$

instead. It expresses that the value achieved when minimizing from $\sigma_2^u(v_2)$ needs to be strictly larger than the value obtained when minimizing from $\sigma_2^l(v_2)$.

Example 59. Reconsider SG in Figure 6.6 and, in particular, the lower and upper strategies from Example 57. With criterion (6.8), we require

$$w_l^\#(\sigma_2^l(v_2^0)) = w_l^\#(v_p^0) = 0 < 1/2 = w_l^\#(v_p^1) = w_l^\#(\sigma_2^u(v_2^0))$$

and therefore admit the strategies.

Criterion (6.8) not only avoids the previously mentioned problems, but it can also efficiently be established in a post-processing step. Roughly speaking, we redirect all choices of the *lower* strategy σ_2^l in player 2 vertices v_2 to the choice of the *upper* player 2 strategy σ_2^u if this does not increase the value in v_2. Formally, we construct the new lower player 2 strategy $\sigma_2^{l\prime}$ as

$$\sigma_2^{l\prime}(v_2) = \begin{cases} \sigma_2^u(v_2) & \text{if } w_l^\#(\sigma_2^u(v_2)) \leq w_l^\#(\sigma_2^u(v_2)) \\ \sigma_2^l(v_2) & \text{otherwise} \end{cases}$$

to avoid unnecessary differences in the player 2 strategies.

An alternative to value iteration is strategy iteration as in [Con90]. It naturally keeps track of strategies, but suffers from similar numerical problems in practice if the underlying linear equation system or LP solvers are imprecise. Using rational arithmetic, strategy iteration provides a means to obtain the desired bounds and strategies in a sound manner.

Refinement for menu-games revolves around the notion of *pivot block* (or vertex) [Wac11]. Intuitively, a pivot vertex is a player 1 vertex of the game at which there is imprecision that is introduced by the abstraction. Formally, we define pivot vertices as follows.

Definition 48 (Pivot Vertex)**.** A player 1 vertex $v_1 \in \mathcal{V}_1$ is a pivot vertex if

$$\sigma_2^l(\sigma_1^l(v_1)) \neq \sigma_2^u(\sigma_1^l(v_1)) \quad \text{or} \quad \sigma_2^l(\sigma_1^u(v_1)) \neq \sigma_2^u(\sigma_1^u(v_1)).$$

That is, for a pivot vertex v_1, player 2 resolves the nondeterminism *differently* in either $\sigma_1^l(v_1)$ or $\sigma_1^u(v_1)$ depending on the optimization direction. Intuitively, as player 2 resolves the nondeterminism introduced by the abstraction, differing choices indicate that the abstraction introduced the imprecision. It is easy to see that there is at least one pivot vertex if the bounds obtained using the abstraction do not coincide.

We remark that the original definition of a pivot vertex [Wac11] is somewhat different.

Definition 49 (Pivot Vertex (as in [Wac11])). A player 1 vertex $v_1 \in \mathcal{V}_1$ is a pivot vertex if $\sigma_2^l(\sigma_1^l(v_1)) \neq \sigma_2^u(\sigma_1^u(v_1))$.

We will detail later why the original definition does not suffice to guarantee a successful refinement of the partition if property (6.7) is not satisfied and stick to our definition of a pivot vertex from now on.

Recall that in our setting, player 2 and probabilistic vertices (other than $v_\perp^p$) are uniquely associated with player 1 and player 2 vertices, respectively. In particular, this means that both player 2 and probabilistic vertices are uniquely associated with an action α and — because the edges of the SPA $\mathfrak{A}$ are assumed to be labeled uniquely — also with an edge e.

Given a pivot vertex v_1 (according to Definition 48), we consider its player 2 successors $v_2^l = \sigma_1^l(v_1)$ and $v_2^u = \sigma_1^u(v_1)$ under both player 1 strategies. According to our definition of a pivot vertex, we have

$$\sigma_2^l(v_2^l) \neq \sigma_2^u(v_2^l) \quad \text{or} \quad \sigma_2^l(v_2^u) \neq \sigma_2^u(v_2^u).$$

As the other case is symmetric, let us assume that $\sigma_2^l(v_2^l) \neq \sigma_2^u(v_2^l)$. Let $v_p^l = \sigma_2^l(v_2^l)$ and $v_p^u = \sigma_2^u(v_2^l)$. If either one of them is equal to $v_\perp^p$ this means that player 2 chose to move to the trap vertex and that there is some state in v_1 that does not have the action associated with v_2^l enabled. The partition can therefore be suitably refined by adding the guard of the corresponding edge e, because this separates states in v_1 in which e is enabled from those where it is not.

If both v_p^l and v_p^u are not $v_\perp^p$, we proceed differently. Since v_p^l and v_p^u are associated with the same edge, they result from the same symbolic probability distribution and assignments to the variables. That is, $v_p^l \neq v_p^u$ implies that there is at least one assignment a and two successor vertices $v', v'' \in succ(v_p^l) \cup succ(v_p^u), v' \neq v''$ such that $v_p^l(a, v') = v_p^u(a, v'') > 0$. In the setting of predicate abstraction, $v' \neq v''$ means that there must be a predicate φ having a different value in the two vertices. Adding $wp(\varphi, a)$ refines the abstraction suitably, because it eliminates the particular choice for player 2 in v_2^l.

Recall that for a player 1 vertex v, $v(i)$ denotes the truth value of the predicate φ_i in v, because it corresponds to a block of the partition. Formally, we define an operator

NewPred responsible for deriving new predicates from a pivot vertex v_1 by

$$\textsc{NewPred}(v_1) = \textsc{NewPred}(v_2^l = \sigma_1^l(v_1)) \cup \textsc{NewPred}(v_2^u = \sigma_1^u(v_1))$$

$$\textsc{NewPred}(v_2 = \langle s, \alpha \rangle) = \begin{cases} \varnothing & \text{if } \sigma_2^l(v_2) = v_2^u(v_2) \\ \{ge\} & \text{if } \sigma_2^l(v_2) = v_\perp^p \vee \sigma_2^u(v_2) = v_\perp^p \\ \{wp(\varphi_i, a)\} & \text{if } v_p^l(a, v') = v_p^u(a, v') > 0 \wedge v'(i) \neq v''(i) \end{cases}$$

where $e \in E$ is the (unique) edge with action α and $v_p^l(a, v'), v_p^u(a, v'') \in \widehat{\mathcal{V}_p}$ are probabilistic vertices extended with the corresponding assignments. NewPred takes a pivot vertex and returns a non-empty set of predicates that can be added to the current set. They refine the partition such that progress can be ensured [Wac11], where progress is to be understood as separating states contained in the pivot vertex that have different lower or upper bounds with respect to the current partition. However, this result implicitly operates under the assumption (6.7) and, as we will now show, fails to hold if the strategy derivation process (like the one in [Wac11]) does not guarantee this condition. For this, we illustrate that Definition 49 of a pivot vertex from [Wac11] is insufficient if the aforementioned property is not satisfied in the sense that $\textsc{NewPred}(v_1)$ may return an empty set of predicates for a pivot vertex when defined as in Definition 49.

Example 60. Reconsider the SG $\mathcal{G}$ and the lower and upper strategy pairs $\langle \sigma_1^l, \sigma_2^l \rangle$ and $\langle \sigma_1^u, \sigma_2^u \rangle$ from Example 57 that do not satisfy Equation (6.7). According to Definition 49, v_1 is a pivot vertex. However, $\textsc{NewPred}(v_1) = \varnothing$, since NewPred cannot find a deviation in any of the two player 2 successors v_2^0 and v_2^1.

Recall that an *interpolant* φ succinctly capture the essence of why the conjunction of a pair of formulae $\langle \varphi_1, \varphi_2 \rangle$ is unsatisfiable (see Section 2.6). As it turns out, this can be leveraged to derive new predicates in our setting. More specifically, they can help eliminating *spurious* pivot vertices. Note that the abstract game may contain reachable blocks (player 1 vertices) such that all contained concrete states of the PA $\mathcal{M}$ are in fact unreachable in $\mathcal{M}$. Naturally, it may be the case that such a block is a pivot vertex and used for the derivation of new predicates. However, we would clearly prefer to introduce predicates that make the pivot vertex unreachable altogether. In general, it is undecidable whether a pivot vertex is reachable or not, so we resort to the approach in [Wac11]. Here, after picking pivot vertex v_1, the most probable path ρ in the DTMCs induced by $\langle \sigma_1^l, \sigma_2^l \rangle$ or $\langle \sigma_1^u, \sigma_2^u \rangle$ is computed. Then, the trace formula φ_ρ for ρ is constructed as in [HWZ08] and its (un)satisfiability intuitively determines whether there exists a concrete path in $\mathcal{M}$ that follows the choices in terms of edge and assignment selection made by the abstract path. If φ_ρ is satisfiable there is such a concrete path and we proceed as presented above. However, if φ_ρ is unsatisfiable it means that the abstraction spuriously introduced the abstract path. Note that there may still be some state in the block v_1 that is reachable, simply via a different (abstract) path. In this case, we split the trace formula φ_ρ at different positions and obtain predicates for the resulting formula pairs via interpolation, which intuitively summarize the reason why the abstract path is spurious.

Finally, we emphasize that the choice of the pivot vertex governs the choice of predicates and therefore also the quality of progress. In general, only pivot blocks that are reachable under $\langle \sigma_1^l, \sigma_2^l \rangle$ or $\langle \sigma_1^u, \sigma_2^u \rangle$ need to be considered, because only they actually explain the deviation. Still, it remains unclear how to determine the pivot vertex that realizes the "best progress" and we resort to reasonable heuristics. A first heuristic is to pick a pivot vertex v_1 that additionally maximizes

$$w_u^\#(v_1) - w_l^\#(v_1), \tag{6.9}$$

because intuitively this pivot vertex causes the largest imprecision. Another approach is to consider the pivot vertex with the minimal distance from the initial vertex v_0. Here, the distance can be either measured in terms of discrete steps or in terms of probability, resulting in the notion of most-probable paths. Finally, the two approaches can be combined. For instance, the deviations according to Equation (6.9) can be weighted with the probabilities of the most-probable paths leading to the respective pivot vertices. This results in choosing a pivot vertex that appears to be both reasonable in terms of the *local* imprecision it causes as well as the *global* effect of this imprecision on the bounds of the initial vertex.

6.4.5 Optimizing the Abstraction Process

We have shown how menu-games can be used in a fully automated abstraction-refinement loop. The result of each iteration of the loop are lower and upper bounds for the target reachability property. In case the bounds are not precise enough, the abstraction is refined and the abstraction process is repeated. While we presented a functional method that can be used to realize the abstraction-refinement loop, a naive implementation will be prohibitively expensive in terms of runtime. In this section, we develop optimizations that speed up the presented approach by orders of magnitude. Here, we focus on the abstraction process entirely, because it is the most specific to menu-based abstraction-refinement. Techniques to speed up the solution of the abstract game are likely independent of the considered abstraction.

Recall that Algorithm 8 computes the abstract menu game by enumerating the solutions of an SMT problem. More specifically, it enumerates the solutions of

(i) the abstract transition constraint $\varphi_e^\#$ for an edge $e \in E$, and

(ii) the abstract expression constraint $\varphi_\eta^\#$ for a Boolean expression $\eta \in Bxp(Var)$.

Since the former is combinatorially more expensive as it encodes not only source but also several successor vertices, we exclusively focus on this part of the abstraction. Parts of the optimizations we will now present are also applicable to the abstract expression constraints, though.

As the following sections revolve around the abstract transition constraint, we explicitly re-state it here for better readability. For an edge $e = \langle \ell, g, \alpha, D \rangle \in E$, the abstract transition constraint (Definition 47) can be simplified to

$$\varphi_e^\# \equiv g \wedge \bigwedge_{i=0}^{n-1} \left(b_i^0 \iff \varphi_i \right) \wedge \bigwedge_{j=1}^{|D|} \bigwedge_{i=0}^{n-1} \left(b_i^j \iff wp(\varphi_i, a_j^D) \right)$$

where

» $\Phi = \{\varphi_0, \ldots, \varphi_{n-1}\}$ is the current set of predicates,

» b_i^j is the jth copy of the Boolean variable b_i that corresponds to predicate φ_i (with the "exception" that $b_i^0 = b_i$), and

» $a_j^D(x)$ is the expression that the jth assignment of the symbolic probability distribution D assigns to the variable x.

Variable Ranges. A straightforward optimization to the enumeration process is the assertion of variable ranges. As it stands, the abstract transition constraint does not constrain the solutions to the domains of the variables. Including the constraints

$$x \in Dom(x) \qquad \text{for every } x \in Var$$

may severely limit the search space of the SMT solver.

Incrementality. Another optimization exploits the internal structure of modern SMT solvers. More concretely, most solvers work *incrementally* in the sense that every solving process derives conflict clauses that essentially capture information that is implied by the original problem and can help in subsequent calls (see Section 2.6). This information can easily be preserved when adding new formulae to the SMT problem, because they are interpreted as being in *conjunction* with the previous formulae and the implications therefore remain valid. Clearly, in our context it is desirable to maintain the solver over successive abstraction steps.

This can easily be achieved by (i) creating a separate Smt solver instance for each edge $e \in E$ and (ii) observing that introducing new predicates only requires adding new conjuncts to the previous abstract transition constraint. To see the latter, let $\varphi^{\#}_{e,\Phi}$ be the abstract transition constraint with respect to the current set of predicates $\Phi = \{\varphi_0, \dots, \varphi_{n-1}\}$. For a new predicate set $\Phi' = \{\varphi_0, \dots, \varphi_{n-1}, \dots, \varphi_{n'}\}$, we can create the new abstract transition constraint $\varphi^{\#}_{e,\Phi'}$ as

$$\varphi^{\#}_{e,\Phi'} \equiv \varphi^{\#}_{e,\Phi} \;\wedge\; \underbrace{\bigwedge_{j=1}^{|D|} \bigwedge_{i=n}^{n'} \Big(b^j_i \iff wp(\varphi_i, a^D_i) \Big)}_{\varphi_{\Phi,\Phi'}}$$

and we can therefore simply add $\varphi_{\Phi,\Phi'}$ to the solver when moving from Φ to Φ'.

Unaffected Edges. When introducing new predicates, it may well happen that the set of variables occurring in the new predicates are disjoint from the variables occurring in a particular edge e (that is, in its guard and assignments as either the written or read variables). In this case, the previous abstraction of e in the form of the MTBDD M_e obtained via AbstractEdge(e) (Algorithm 8) can be extended with the information that the value of the predicate is the same in all source and target blocks of the transitions associated with e.

Relevant Predicates. Generalizing the concept of unaffected edges, [Wac11] sketches that one can reduce the size and number of solutions of the abstract transition constraint $\varphi^{\#}_e$ by reducing the number of considered predicates. For example, in order to determine in which blocks an edge is enabled, it suffices to consider the predicates that share a variable with the guard. Similarly, to determine the validity of predicates in the successor block of an assignment a of an edge, one has to consider all predicates

(i) that contain a variable that is being assigned (a potentially different value) in a, because these predicates might change value,

(ii) that share a variable with any of the expressions assigned to a variable, because they influence the values of aforementioned predicates.

[Wac11] then goes on to assemble two sets of predicate indices G_1 and G_2 where

G_1 is equal to the predicates in (ii) *plus* the predicates that share a variable with the guard, and

G_2 corresponds to the union of the predicates in (i) for all assignments in the support of the symbolic probability distribution.

The claim now is that the abstract transition constraint can be simplified to

$$g \wedge \bigwedge_{i \in G_1} \left(b_i^0 \iff \varphi_i\right) \wedge \bigwedge_{j=1}^{|D|} \bigwedge_{i \in G_2} \left(b_i^j \iff wp(\varphi_i, a_j^D)\right)$$

without losing information in the sense that the solutions of the simplified abstract transition constraint can be extended with the missing predicates retaining their value to obtain the original solutions. However, this is not true as the following example shows.

Example 61. Consider an edge $e = \langle \ell, true, \alpha, D \rangle$ of an SPA with a single location ℓ where D given as

$$D(a, \ell) = \begin{cases} 1 & \text{if } a = a_\perp[y \mapsto 1] \\ 0 & \text{otherwise.} \end{cases}$$

Furthermore, we have the predicate set

$$\Phi = \left\{ \underbrace{x = 1}_{\varphi_0}, \underbrace{x + y = 2}_{\varphi_1} \right\}$$

and determine the sets G_1 and G_2 according to the above definitions as

$$G_1 = \varnothing \quad \text{and} \quad G_2 = \{x + y = 2\} = \{\varphi_1\}\,.$$

We construct the simplified abstract transition constraint according to [Wac11] as

$$\varphi_e^\# \equiv \left(b_1^1 \iff x + 1 = 2\right).$$

One solution v of this constraint is

$$v(b_1^1) = 0.$$

Extending this with the predicate identity for φ_0 yields that the block characterized by

$$\Phi_1 = \left\{ \underbrace{x = 1}_{\varphi_0}, \underbrace{x + y \neq 2}_{\neg\varphi_1} \right\}$$

has a successor block characterized by

$$\Phi_2 = \left\{ \underbrace{x = 1}_{\varphi_0}, \underbrace{x + y \neq 2}_{\neg\varphi_2} \right\}.$$

However, this clearly should not be the case since (i) x is 1 before and after the transition, (ii) y is 1 after the transition as dictated by $a_\perp[y \mapsto 1]$, and consequently (iii) the predicate $\varphi_2 = (x + y = 2)$ needs to hold after taking edge e. It can be easily verified that the full abstract transition constraint does not permit this solution.

This example shows that it is insufficient to consider G_1 and G_2 as above when the goal is to obtain the exact menu-based abstraction. In fact, the simplification does not only introduce wrong transitions, it may even add transitions to blocks that are inconsistent and therefore do not contain any state. In principle, there are two angles to approach this problem.

The first one is that it might be acceptable to obtain an *over-approximation* of the menu-game that contains superfluous behavior or even inconsistent blocks. However, it is not clear whether this still guarantees that for finite models the abstraction-refinement process eventually terminates with a conclusive answers as the inconsistent blocks potentially introduce imprecision indefinitely.

The other option is to reduce the number of predicates less aggressively. By refining the sets G_1 and G_2 we seek to rule out the spurious behavior and inconsistent successor blocks. For this, we need the notion of when two predicates are indirectly related. We call two predicates φ, φ' indirectly related w.r.t. Φ, written $\varphi \equiv_\Phi \varphi'$, if they are in the same equivalence class of the transitive closure of the relation that connects all predicates in Φ that share any variables. The intuitive reason for the problem described above is that the definition for G_1 decouples predicates that are indirectly related by using shared variables. Similarly, we call a variable x indirectly related to the predicate φ' w.r.t. Φ, written $x \equiv_\Phi \varphi'$, if there is some predicate $\varphi \in \Phi$ containing x such that $\varphi \equiv_\Phi \varphi'$.

Finally, we let two variables x, x' be indirectly related w.r.t. Φ, written $x \equiv_\Phi x'$, if there exists a predicate $\varphi' \in \Phi$ containing x' such that $x \equiv_\Phi \varphi'$.

We refine the notion of [Wac11] by considering the following sets of relevant predicates:

G_1' contains the predicates that are in G_1 or are *indirectly related* to a variable that is assigned to by a_j^D for any j,

$G_2(j)$ contains the predicates that contain some variable that is assigned to by a_j^D.

Now, we can simplify the abstract transition constraint to

$$\varphi^\#_{e,\mathrm{rel}} = g \wedge \bigwedge_{i \in G_1'} \left(b_i^0 \iff \varphi_i\right) \wedge \bigwedge_{j=1}^{|D|} \bigwedge_{i \in G_2(j)} \left(b_i^j \iff wp(\varphi_i, a_j^D)\right)$$

and obtain the original solutions by preserving the values of the remaining predicates along the transition. Again, strictly speaking this is not true as the above construction may also introduce inconsistent blocks. However, it guarantees that no consistent block has an inconsistent successor block. Performing a reachability analysis starting from the initial states as suggested in Section 6.4.2.3 therefore removes all inconsistent blocks and arrives at precisely the same game as without the optimization.

Lemma 7 (Correctness of Relevant Predicates Optimization)**.** *Let $e = \langle \ell, g, \alpha, D\rangle \in E$ be an edge and $v \in Val(\mathfrak{B}^{\leq |D|})$ be a variable valuation such that the block represented by $v|_{\mathfrak{B}}$ is consistent, i. e.*

$$\bigwedge_{i=0}^{n-1} \left(v(b_i) = 1 \iff \varphi_i\right)$$

is satisfiable. Then

$$v \models \varphi^\#_e \iff v \models \varphi^\#_{e,\mathrm{rel}}.$$

Edge Decomposition. The *relevant predicates* optimization hints at partitioning the predicates in such a way that predicates that may influence each other's truth values belong to the same predicate class. In the following, we consider two variables $x, x' \in Var$ to be related by the edge $e = \langle g, \alpha, D\rangle$, written $x \equiv^*_{e,\Phi} x'$ if they are in the transitive and reflexive closure of the relation $\equiv_{e,\Phi}$ where $x \equiv_{e,\Phi} x'$ if

» the variables are already indirectly related w.r.t. Φ, i. e. $x \equiv_\Phi x'$, or

» both variables are contained in an expression assigned to a variable x'' by some assignment a_j^D along e, i. e. there exists j and x'' such that

$$x \in Var\left(a_j^D(x'')\right) \text{ and } x' \in Var\left(a_j^D(x'')\right), \text{ or}$$

» the variables are connected via an assignment a_j^D, i. e. there exists j such that

$$x' \in Var\left(a_j^D(x)\right) \text{ or } x \in Var\left(a_j^D(x')\right).$$

Given the relation $\equiv_{e,\Phi}^*$, the edge can be decomposed into sub-edges that essentially correspond to restricting the edge e to the different equivalence classes of $\equiv_{e,\Phi}^*$. Intuitively, this is because the equivalence classes are independent and the solutions of the abstract transition constraint correspond to the Cartesian product of the solutions of the abstract transition constraints for the sub-edges. This decomposition reduces the number of solutions that need to be enumerated from the product of the solutions for the individual equivalence classes to their sum. We remark that this optimization can be combined with the relevant predicates optimization by first decomposing the edge and then, for the subedges restricting to the relevant predicates.

Expression Decomposition. Both, the relevant predicates and the edge decomposition optimization are strongly influenced by the relation $\equiv_\Phi$ that transitively relates variables that appear in predicates sharing variables. The optimizations tend to perform the best if the partitioning is *finer* and only few variables are related, because then the number of solutions that need to be enumerated decreases drastically. In turn, this relation is strongly influenced by which predicates are used to refine the abstraction. In particular, composed predicates that are, for example, the conjunction or disjunction of other subpredicates, may be added to the current predicate set. To maintain a finer partitioning induced by $\equiv_\Phi$, these composite predicates may be split before adding them to the set of predicates. However, we remark that in general this may lead to a finer partition Π of the states and therefore impacts the sizes and solution times of the abstract games adversarially.

6.5 Evaluation

We have prototypically implemented the fully automated game-based abstraction approach in the framework of Storm (see Chapter 7). For the enumeration of feasible solutions of the abstract transition constraints we rely on the Smt solver MathSat [Cim+13], because it provides dedicated efficient AllSat enumeration as well as interpolation. Our prototypical implementation can abstract models given in both the **PRISM** language and **JANI** (see Chapter 3), implements all optimizations mentioned before Section 6.4.5 and is highly configurable. More specifically, there are various options that influence the abstraction-refinement process. Among other things, the user can select

» the technique used to solve the abstract games (policy iteration or value iteration),

» whether the games are solved symbolically (DD) or using sparse matrices,

» which DD library is used for the representation of the abstract game (CUDD [Som] or Sylvan [Dij16]),

» whether or not all guards of edges are added initially (which avoids dealing with the special bottom vertices),

» whether or not the expression characterizing the initial states is added as a predicate, and

» which data type (and precision) to use for the computation (floating point or rational numbers).

In particular, it is possible to obtain sound lower and upper bounds in practice when using rational numbers and policy iteration as the solution technique. Because of the inherent rounding errors in floating point arithmetic [Wim+08] and the problems related to convergence of value iteration [HM14; Bai+17b; QK18; Kel+18], this is otherwise *not* guaranteed.

In our evaluation, we choose to use the following configuration:

» games are solved using policy iteration and floating point arithmetic,

» the MTBDD representation of the abstract game is realized via CUDD,

» games are solved using a sparse matrix representation,

» all guards are added as predicates, and

» the expression characterizing the initial states is added as a predicate.

Let us justify these settings. First of all, policy iteration avoids some precision issues of value iteration and simultaneously can lead to strategies that only differ at crucial places, which in turn positively influences the progress achieved via refinement. Secondly, it is well known that sparse representations typically outperform symbolic ones as long as they fit into in memory. Finally, we observe that adding more predicates (such as the guards and initial expression) increases abstraction times, but not only yields tighter bounds but also may result in *smaller* abstractions, a phenomenon that we will revisit in the course of our evaluation. Ultimately, it is rooted in the fact that *additional predicates might constrain the reachable state space of the game*, which is especially the case for predicates obtained via interpolation. Consequently, lacking those predicates may produce a potentially large number of unreachable player 1 vertices and since the goal of the abstraction process is to derive interesting probability bounds on *small* abstractions, we somewhat counterintuitively have to add *more* predicates.

Benchmarks. To evaluate the effectivity of game-based abstraction using menu-games, we consider seven benchmark models and several properties on each of these models. Four of these were considered in the context of game-based abstraction in [Wac11]. Two of the three models we additionally consider (`coin` and `zeroconf`) are taken from Prism's benchmark suite. Finally, the coupon collector example is a modified version of the model in the repository of **JANI** examples available at `https://github.com/ahartmanns/jani-models/`. Both, the models and the properties we used are detailed in Appendix E. Except for one property on the `coin` model, all properties are quantitative in nature and ask for minimal and maximal reachability probabilities. We continue the abstraction-refinement process until the obtained lower and upper bounds for the initial vertex achieve a *relative* precision of $\epsilon_a = 10^{-3}$, i. e.

$$\frac{u - \ell}{u + \ell} \leq 10^{-3}$$

where u is the upper and ℓ the lower bound, respectively. This criterion ensures that for very small probabilities a sufficient precision is attained. We let the numerical solution techniques involved in the solution step use a relative precision of $\epsilon_s = 10^{-6}$. In general, the precision of the solution process ϵ_s should be significantly higher than the precision to be achieved by the abstraction process ϵ_a, because the latter depends crucially on meaningful strategies to drive the refinement. Using a coarser solution precision may result in worse bounds and worse strategies and therefore in worse or even no progress.

Effectiveness of the Abstraction. As a first step, we consider the sizes of the games when the abstraction is fine enough in the sense of the previously mentioned criteria

		original		abstraction	
model	instance	states	transitions	states	transitions
brp	(64, 5, p_1)	4936	6659	4924	6647
	(64, 5, p_4)	5192	6915	21	27
	(≥ 16, 5, p_1)	∞	∞	4220	5715
	(≥ 16, 5, p_4)	∞	∞	21	27
coin	(4, 6, c_1)	63616	213472	63616	214368
	(4, 6, c_2)	63616	213472	31213	124391
	(6, 2, c_1)	1258240	6236736	≥ 1237760	≥ 6211544
	(6, 2, c_2)	1258240	6236736	438019	2579751
csma	(3)	4314	5569	3814	5067
	(4)	11563	15355	13662	18520
	(5)	31370	42381	35007	47894
swp	(gp)	∞	∞	82655	320323
	(to)	∞	∞	3	4
wlan	(7, min_3)	1299806	2688016	665	1270
	(7, max_3)	1299806	2688016	4763	5455
	(7, max_6)	≥ 1299806	≥ 2688016	68345	163887
	(8, min_6)	1299806	2688016	665	1270
	(8, max_3)	1299806	2688016	4763	9466
	(8, max_6)	≥ 1299806	≥ 2688016	68345	163887
zeroconf	(16, min)	5009561	11306728	164677	397242
	(16, max)	5009561	11306728	≥ 2032954	≥ 5203808
	(20, min)	5811209	13110562	210517	506874
	(20, max)	5811209	13110562	≥ 2615700	≥ 6772944
coupon	(5, 3, r)	∞	∞	99097	161241
	(5, 3, c)	∞	∞	56799	149379
	(7, 3, r)	∞	∞	1281653	2309021
	(7, 3, c)	∞	∞	1125884	3305936

Table 6.1: The sizes of the models and their (final) game-based abstraction.

(relative precision $\epsilon_a = 10^{-3}$). Table 6.1 lists the benchmark instances and their original size. Note that the sizes of the model depend on the property, because Storm only explores the model as long as necessary. In particular, if a reachability property is given, the states that are only reachable from the initial state via target states do not influence the probability from the initial state and are therefore directly omitted. Next to the original model size, we give the size of the final abstract game in terms of player 1 states and transitions. If the experiments timed out and we were unable to give the sizes of models or abstractions, we indicate a lower bound based on smaller model instances or the current state of the abstraction-refinement process.

We observe game-based abstraction refinement is able to obtain a result for all infinite models using a finite abstraction. In the extreme case of the `swp(to)` instance, the final abstract game only has 3 states and 4 transitions. However, as we already mentioned, the intermediate games may be *larger* and in the case of `swp(to)` the largest intermediate game has 9239 player 1 states and 44395 transitions.

The results show that there are finite models for which the abstraction does not yield reductions. We see this behaviour, for example, on the `brp(64,5,p`$_1$`)` instance, for all `coin` instances with property c_2 and all `csma` instances. For the latter, the abstraction is even *larger* than the original model. As previously mentioned, this is a result of the abstraction that oblivious of unreachable concrete states in abstract vertices.

However, for all of the `wlan` and some of the `zeroconf` instances, the abstraction is able to determine the reachability probabilities on a much smaller abstract model. The `zeroconf` case study further emphasizes that the game-based abstraction process is property driven. For one of the two considered properties (min), the abstraction is more than one order of magnitude smaller than the original model, whereas for the other property (max) the abstraction needs to be much finer to reach the desired precision.

Runtimes and Comparison with Regular Verification. To evaluate whether the abstraction can also yield improvements with respect to time and memory requirements of the verification tasks, we compare our prototype to the performance of Storm's other verification engines. More specifically, we use Storm's dd and hybrid engines, because in spirit they are the closest to game-based abstraction in the sense that both build an MTBDD-based representation of the system prior to verification to save memory. Note that game-based abstraction is the only of these techniques that is capable of treating infinite models.

Table 6.2 shows a comparison of these two standard engines with our prototype in which all runtimes are given in seconds and all runs were limited to one hour of runtime and 16GB of RAM. As a measure of quality of the abstraction, we additionally give the (absolute) gap between the lower and upper bounds obtained via the abstraction.

We observe a diverse picture. For the brp, half of the coin and all csma instances, the regular verification techniques outperform the abstraction by orders of magnitude. These are also the instances where the abstraction did not achieve any state space reductions. In the case that there was some but not a great reduction in size as for coin instances with property c_2, the runtimes of game-based abstraction approach those of the standard techniques.

Even though the abstraction could produce a significantly smaller state space for the zeroconf models with property min, this gain is only translated to a minor gain in runtimes. This is due to the relatively costly computation of the abstraction together with the number of refinements, which we will shed more light on in course of the evaluation. However, for the wlan case study, the abstraction is small and outperforms the competing engines by far.

Finally, we want to remark that for many instances the bounds were *precise* in the sense that the lower and upper bounds coincided. Also, even though the abstraction timed out on some of the zeroconf models, the bounds are still rather precise and valuable. For both instances, the absolute gap between the lower and upper bound was smaller than 10^{-4} after one hour. To get a more nuanced picture for these instances, we looked at the quality of the bounds that were obtained up until the point where the *best* of the other standard engines (hybrid) returned the answer. In both cases, the gap was lower than $4 \cdot 10^{-3}$ at this point. Consequently, for a lower precision, the game-based abstraction could have beaten the established engines in terms of runtime.

Time-breakdown. To get a clearer picture regarding the distribution of (runtime) cost for game-based abstraction, we measured the contribution of major building blocks of the abstraction-refinement process to the overall runtime. More specifically, we distinguish the three phases:

» the extraction of the abstract game from the symbolic model (abstraction),

» the solution of the abstract game to obtain lower and upper bounds and the strategies realizing them including the translation from the DD-based game representation to a sparse one (solution), and

» the analysis of the strategies and derivation of new predicates (refinement).

		dd	hybrid	abstraction	
model	instance	time	time	time	gap
brp	(64, 5, p_1)	12.04	0.69	83.74	1.6×10^{-14}
	(64, 5, p_4)	0.21	0.22	2.31	0
	($\geq$ 16, 5, p_1)	-	-	496.97	0
	($\geq$ 16, 5, p_4)	-	-	2.36	0
coin	(4, 6, c_1)	0.21	0.21	586.46	0
	(4, 6, c_2)	783.85	13.73	323.47	0
	(6, 2, c_1)	0.54	0.50	TO	7.7×10^{-1}
	(6, 2, c_2)	872.64	67.28	924.58	6.6×10^{-12}
csma	(3)	0.82	0.29	2594.02	0
	(4)	0.42	1.25	1753.75	1.5×10^{-3}
	(5)	1.88	0.89	2712.22	2.0×10^{-3}
swp	(gp)	-	-	328.07	2.0×10^{-8}
	(to)	-	-	9.27	0
wlan	(7, min_3)	2932.78	2831.63	12.15	0
	(7, max_3)	1503.24	1338.24	109.74	0
	(7, max_6)	TO	TO	822.54	0
	(8, min_3)	3214.85	TO	15.32	0
	(8, max_3)	1820.17	1804.81	127.09	0
	(8, max_6)	TO	TO	886.20	0
zeroconf	(16, min)	TO	277.49	171.39	0
	(16, max)	TO	343.00	TO	7.6×10^{-5}
	(20, min)	TO	323.71	211.52	0
	(20, max)	TO	442.12	TO	7.8×10^{-5}
coupon	(5, 3, r)	-	-	10.08	0
	(5, 3, c)	-	-	21.70	0
	(7, 3, r)	-	-	234.21	0
	(7, 3, c)	-	-	708.07	0

Table 6.2: Game-based abstraction versus standard probabilistic model checking.

model	instance	abstraction	solution	refinement
brp	(64, 5, p_1)	50.8%	35.2%	11.7%
coin	(4, 6, c_2)	36.6%	60.0%	2.8%
csma	(5)	86.4%	3.4%	10.1%
wlan	(8, max_6)	78.9%	8.0%	3.9%
zeroconf	(20, min)	77.2%	19.7%	2.3%

Table 6.3: Runtime distribution of game-based abstraction.

In Table 6.3 we list the percentages for each of those phases for selected, representative models where the remaining time is spent on setup and auxiliary operations.

As the data shows, the *abstraction tends to be the most expensive task while refinement time typically contributes very little to the running times.* Only if the underlying model is already numerically harder to solve (in our case for `brp`, `coin` and `zeroconf`), the time spent on solving the abstract games becomes noticeably larger.

Comparison with Pass. To the best of our knowledge, Pass [Wac11] is the only other publicly available tool for the automated game-based abstraction of **PRISM** (or **JANI**) models. Pass was developed over several years and contains sophisticated optimizations that go beyond what we have described and implemented in our prototype. We are thankful to the authors that they provided us with the **C++** source code of the tool, which is neither publicly available nor maintained any more. Sadly, we were unable to build the tool from source using current compilers, due to changes in the **C++** standard and the used libraries. However, the website of Pass offers a binary that works on 64 bit Linux machines and we used this version to compare the two implementations.

For the comparison with Pass, we consider all mentioned benchmark models except for the `coupon` case study, which is encoded in **JANI** and therefore not supported by Pass.

Table 6.4 displays the results of our experiments. For both our implementation ("Storm") and Pass we give the number of refinements made ("#ref"), the number of predicates that were used in the final abstraction ("#pred") and the runtime. Entries that are marked with a star indicate a wrong result.

Overall, we find that both implementations have advantages on specific models. For example, Pass outperforms our implementation for the `wlan` instances significantly. Conversely, for the `zeroconf` instances, our prototype tends to obtain the answer more

		STORM			PASS		
model	instance	#ref	#pred	time	#ref	#pred	time
`brp`	(64, 5, p_1)	65	100	83.74	≥ 4	≥ 38	TO
	(64, 5, p_4)	4	39	2.31	5	36	1.41
	(≥ 16, 5, p_1)	21	57	496.97	0*/≥3	29*/≥32	0.47*/TO
	(≥ 16, 5, p_4)	4	39	2.36	0*/≥7	31*/≥45	0.49*/TO
`coin`	(4, 6, c_1)	49	75	586.46	15*	48*	50.14*
	(4, 6, c_2)	45	73	323.47	36	78	2088.05
	(6, 2, c_1)	≥ 24	≥ 61	TO	15*	56*	2677.00*
	(6, 2, c_2)	23	61	924.58	≥ 7	≥ 53	TO
`csma`	(3)	102	138	2288.77	≥ 48	≥ 136	TO
	(4)	104	152	1910.36	≥ 33	≥ 97	TO
	(5)	127	178	2760.38	≥ 50	≥ 162	TO
`swp`	(gp)	16	42	328.07	6	48	24.60
	(to)	12	40	9.27	7	36	3.97
`wlan`	(7, min_3)	0	71	12.15	0	78	5.68
	(7, max_3)	30	101	109.74	4	112	24.00
	(7, max_6)	58	129	822.54	2	140	28.39
	(8, min_3)	0	73	15.32	0	80	7.53
	(8, max_3)	30	103	127.09	3	113	23.01
	(8, max_6)	58	131	886.20	2	142	28.95
`z'conf`	(16, min)	15	61	171.39	≥ 30	≥ 104	TO
	(16, max)	≥ 28	≥ 76	TO	≥ 4	≥ 99	MO
	(20, min)	19	65	211.52	≥ 20	≥ 110	TO
	(20, max)	≥ 31	79	TO	≥ 7	≥ 110	MO

* Marked entries indicate a wrong result.

Table 6.4: Runtime comparison of STORM's game-based abstraction with PASS.

		Storm	Pass
model	instance	gap	gap
zeroconf	(16, max)	7.6×10^{-5}	2.7×10^{-3}
	(20, max)	7.8×10^{-5}	1.4×10^{-3}

Table 6.5: Interval gaps for instances that did not complete in time.

quickly. Since both tools are not able to completely solve the `zeroconf` instances with the *max* property, we give the absolute gaps between the most precise lower and upper bounds obtained within the resource limit in Table 6.5. As the data shows, the bounds obtained with our prototype are roughly two orders of magnitude more precise.

We want to say a few words about the entries for Pass that are marked as being incorrect in the table. There seem to be two separate problems. For the two affected `coin` instances, it seems to be a simple bug in the logic. Having derived the probability interval $[4.5 \times 10^{-1}, 1]$ for the (minimal) reachability probability, Pass incorrectly concludes that the probability is less than 1 for some initial state. For the `brp` instances, we observe that Pass returns different results based on how the initial states of the models are specified even though the two ways are equivalent. Using an explicit initial construct in the Prism program yields an incorrect result after a very short time while using initial values for the variables results in a time-out.

In general, we notice that Pass tends to need fewer refinements, because its predicate discovery strategy is more aggressive in the sense that often several predicates are added at once. This is in contrast to our prototype that adds predicates more conservatively. For `wlan`, adding more predicates does not negatively impact the size of the (reachable part of the) abstraction and quickly results in tight bound. In contrast, for the `zeroconf` case study, the abstraction tends to grow too quickly when too many predicates are added, which results in an out-of-memory error for Pass. From this, we conclude that *the predicate synthesis strategy and the quality of the synthesized predicates is the major factor for the performance of game-based abstraction-refinement.* To more closely examine this, we added an option to our prototype to inject user-specified predicates. We isolated the predicates that were discovered by Pass for selected benchmark instances and injected them into our prototype. The results of this predicate injection are shown in Table 6.6 where we did not add the predicates related to the initial expression to more closely match the behavior of Pass.

We see that the used predicates strongly influence the performance. Overall, our proto-

		Storm (injected)			Pass		
model	instance	#ref	#pred	time	#ref	#pred	time
`wlan`	(7, $\max_3$)	4	104	30.19	4	112	24.00
	(7, $\max_6$)	2	131	53.25	2	140	28.39
	(8, $\max_3$)	3	106	35.24	3	113	23.01
	(8, $\max_6$)	2	135	63.26	2	142	28.95
`zeroconf`	(16, min)	16	69	1164.44	≥ 30	≥ 104	TO
	(16, max)	≥ 4	89	MO	≥ 4	≥ 99	MO
	(20, min)	13	72	762.93	≥ 20	≥ 110	TO
	(20, max)	≥ 5	≥ 80	MO	≥ 7	≥ 110	MO

Table 6.6: Effect of injecting the predicates obtained by Pass on our prototype.

type performs very similar to Pass when using the same predicates. In particular, on the `wlan` instances, the predicate injection improves the running times and number of refinements whereas for the `zeroconf` instances the performance degrades in comparison to the previous strategy of Storm. We remark that for the `wlan` instances it may be unexpected that Storm uses fewer predicates even though the predicates were injected from Pass. However, this is simply due to our prototype eliminating equivalent predicates where Pass does not do this.

Surprisingly, for the min instances of `zeroconf`, Storm terminates after *fewer* iterations than Pass even though the tools use the same predicates. A look at the sizes of the abstractions reveals that Pass computes a larger abstraction using the same set of predicates. Our guess is that this is due to the presence of spurious behavior and inconsistent blocks (see Section 6.4.5) in the abstraction, which would also explain the less precise bounds obtained by Pass on the same abstraction.

Evolution of Progress. We now move on to analyze the progress of the refinement over successive iterations. More specifically, we consider the evolution of the obtained bounds together with the size of the abstraction in terms of player 1 vertices. For this, we draw figures that plot the progress of the bounds (left y-axis, lower bound in blue, upper bound in red) and the size of the abstraction (right y-axis, plot in green). In Figure 6.8, we observe that for the `coin(6,2,c2)` instance, the size of the abstraction grows roughly linearly, but the bounds only become non-trivial in iteration 13. While

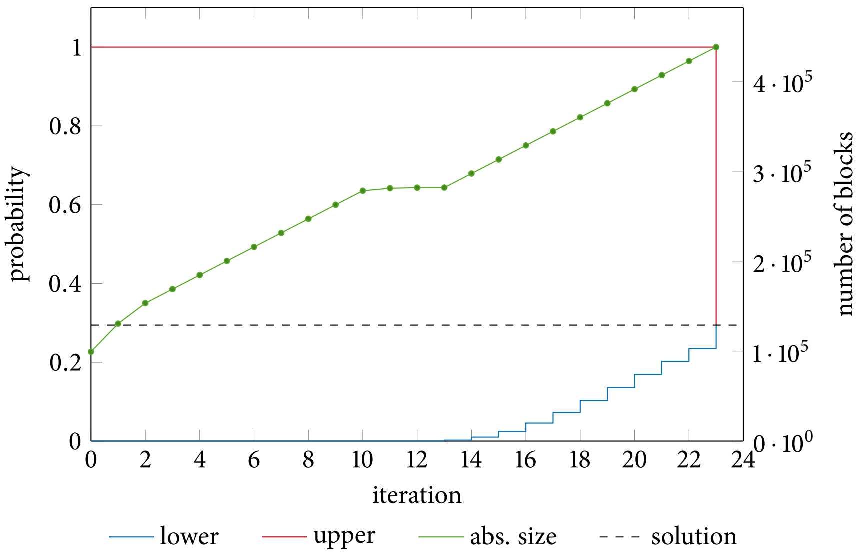


Figure 6.8: `coin(6,2,c`$_2$`)`

the lower bound then steadily approaches the solution, the upper bound is trivial until the very last iteration.

For `wlan(8,10,max`$_3$`)` (see Figure 6.9) the size of the abstraction grows uniformly until iteration 27, but then predicates are found that prune parts of the abstract game and make the bounds sufficiently precise. From the very start, the bounds improve in almost every iteration, but in small steps.

This is in contrast to the progress made on the `swp(gp)` instance, shown in Figure 6.10. Here, only few iterations achieve tighter bounds, but if progress is made it is substantial. Also, we see that the size of the abstraction fluctuates and is not directly correlated with obtaining better bounds. Finally, we consider the `zeroconf(16,12k,max)`. Here, the abstraction only grows moderately until the bounds are already rather precise (in iteration 16). However, to reach the desired precision, more than 13 additional refinements are necessary, which increases the size of abstract game considerably. This exemplifies that the abstraction process can potentially be much cheaper when less precision is required.

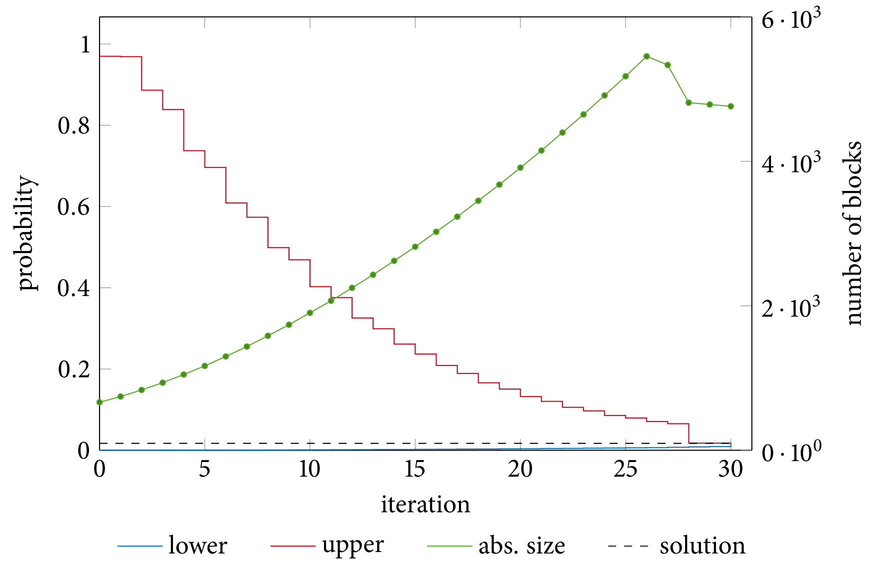


Figure 6.9: `wlan(8,max`$_3$`)`

Soundness. The implementations, in Storm and Pass both suffer from numerical imprecision and are ultimately unsound. This is for two reasons. One is that the rounding that is necessitated by floating point arithmetic may inadmissibly influence the values [Wim+08]. The second is that the numerical techniques typically only achieve a certain precision and, in the case of value iteration, do not give any guarantees regarding the quality of the solution [HM14; Bai+17b; QK18; Kel+18].

Consider, for example the PA $\mathcal{M}(p)$ whose transition probabilities depend on the parameter p shown in Figure 6.12. The reachability probability in $\mathcal{M}(p)$ with respect to the target state s_1 is 1 for all $p \in [0,1)$. Computing the reachability probability for the target state s_1 in $\mathcal{M}(10^{-4})$, Pass returns the interval $[0,0]$ as the result, which is clearly incorrect. Even for $\mathcal{M}(10^{-3})$, Pass cannot determine the probability to be 1 but rather returns $[6.3 \times 10^{-1}, 8.6 \times 10^{-1}]$ and that the result is inconclusive. On this particular example, Storm returns the correct result for all values of $p \in [0,1)$, because unlike Pass it features the precise (probability-independent) computation of states with probability 0 and 1, respectively (see Section 6.4.3.1). Still, it is not difficult to construct examples for which Storm will also fail to derive the correct result because of numerical imprecisions when using floating point arithmetic.

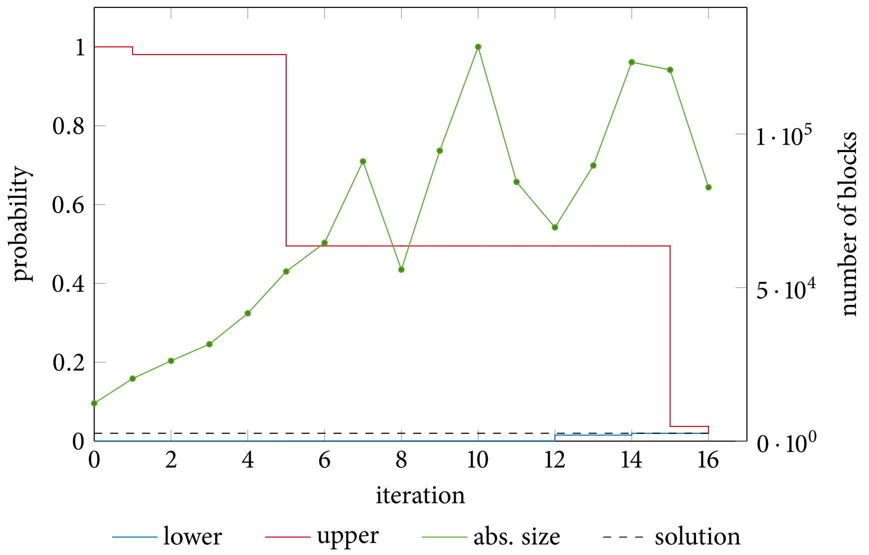


Figure 6.10: `swp(gp)`

An advantageous property of our prototype is that it is highly configurable. In particular, we can solve the abstract games in each iteration using policy iteration and a linear equation solver using rational arithmetic that guarantees exact results. Together, this results in sound bounds for reachability probabilities. To the best of our knowledge, our prototype is the only tool capable of performing abstraction-refinement for potentially infinite PA while obtaining sound bounds for both minimal and maximal reachability probabilities. We therefore compare this configuration of our prototype with Storm's hybrid engine set to produce exact results. Note that strictly speaking the two approaches do not give the same result as our prototype does not necessarily produce an exact answer but rather returns *bounds* that are exact and sound in the sense that the actual value is guaranteed to lie in the derived interval. Since the abstract game is built as an MTBDD and Storm only supports rational numbers in MTBDDs when using the Sylvan library, we switch from CUDD to the latter for these experiments. Note that this affects not only the abstraction but also true for the hybrid engine. As Sylvan is built for multi-threading applications, we let Sylvan use 4 threads in all experiments.

Table 6.7 shows the measured (wall-clock) running times of the aforementioned techniques. As before, it also lists the gap between the last lower and upper bounds that

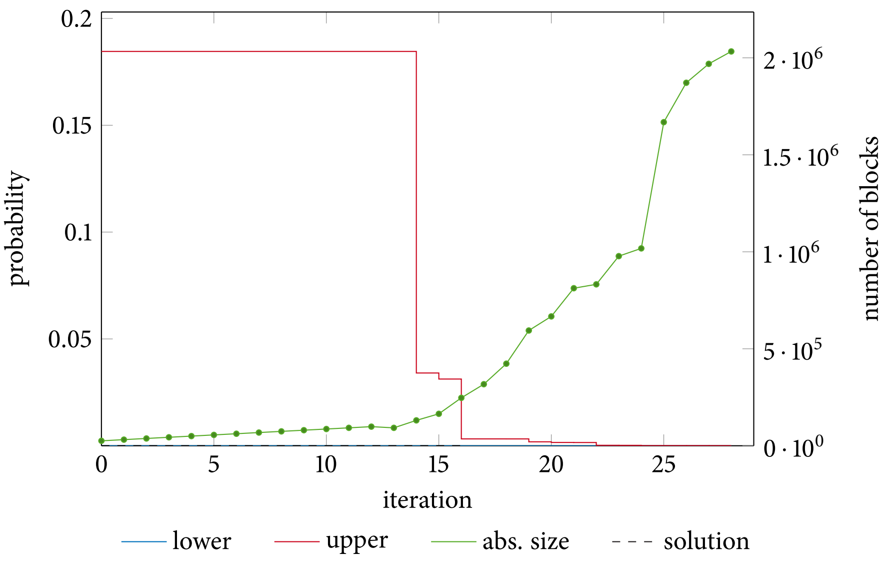


Figure 6.11: `zeroconf(16,max)`

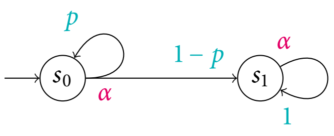


Figure 6.12: A problematic PA.

could be derived within the time and memory limits for the abstraction-refinement approach. Overall, the picture is similar to the floating point case. That is, for some finite models, abstraction-refinement takes significantly longer. In the case of `coin(6,2,c`$_1$`)` it even runs out of memory while the hybrid engine can solve the model very quickly. This is because c_1 is a qualitative property that can be solved without considering the probabilities. In contrast, game-based abstraction needs to repeatedly solve quantitative problems on the involved abstract games, because one of the bounds (the lower bound) is quantitative until the abstraction is precise enough. Finally, we obtain gains for the same models, `wlan` and `zeroconf`, as in the floating point setting. In particular, we are

		hybrid (rational)	abstraction (rational)	
model	instance	time	time	gap
brp	$(64, 5, p_1)$	2.47	165.86	0
	$(64, 5, p_2)$	1.57	5.05	0
	$(\geq 16, 5, p_1)$	-	18.77	0
	$(\geq 16, 5, p_4)$	-	4.34	0
coin	$(4, 6, c_1)$	1.68	813.30	0
	$(4, 6, c_2)$	10.49	415.81	0
	$(6, 2, c_1)$	2.83	MO	1
	$(6, 2, c_2)$	MO	MO	9.8×10^{-1}
csma	(3)	2.28	2687.80	0
	(4)	2.88	2247.80	0
	(5)	4.42	3246.96	2.0×10^{-3}
swp	(gp)	-	791.37	7.0×10^{-6}
	(to)	-	29.33	0
wlan	$(7, \text{min}_3)$	894.49	17.50	0
	$(7, \text{max}_3)$	426.32	139.64	0
	$(7, \text{max}_6)$	MO	967.85	0
	$(8, \text{min}_3)$	1705.59	20.66	0
	$(8, \text{max}_3)$	648.35	149.60	0
	$(8, \text{max}_6)$	MO	1075.99	0
zeroconf	(16, min)	MO	MO	4.8×10^{-15}
	(16, max)	MO	MO	3.1×10^{-2}
	(20, min)	MO	MO	2.7×10^{-17}
	(20, max)	MO	TO	3.1×10^{-2}
coupon	(5, 3, r)	-	54.58	0
	(5, 3, c)	-	268.04	0
	(7, 3, r)	-	MO	1
	(7, 3, c)	-	MO	1

Table 6.7: Abstraction-ref. vs. standard prob. model checking (rational arithmetic).

able to obtain rather precise bounds for the `zeroconf` instances despite finally running out of resources.

Fully Symbolic Abstraction-Refinement. Before concluding, we want to briefly review *fully symbolic* abstraction-refinement. As previously mentioned, our prototype can be configured such that both the construction and the solution of the abstract games is done fully symbolically (on DDs). However, our evaluation focused exclusively on a hybrid approach that first builds the abstract games using MTBDDs, but then transforms them to an explicit representation prior to the numerical analysis. In a nutshell, we observe that in its current state, fully DD-based game-based abstraction refinement is inferior to the hybrid approach on all models. It is to be expected that in terms of runtime MTBDD-based numerical analysis does not outperform the analysis on an explicit representation as long as the latter fits into memory. Therefore, the only fundamental gain is possible in terms of memory requirements. However, in our prototype this is currently rarely the case. One reason for this is that Storm currently only supports value iteration for symbolic games and not policy iteration. In our experiments, we see that value iteration tends to produce less decisive predicates than policy iteration, which results in more refinement operations and therefore not only longer runtimes but also larger games. The latter then diminishes the potential memory savings. A second reason is that symbolic data structures benefit from having symmetry in the model, but in some sense the ultimate goal of the abstraction is to detect and remove these symmetries. Finally, for the considered models, the abstract games are small enough to fit into memory (see Table 6.1). However, this is due to the model selection, which is mostly based on Pass's benchmark models and it is still conceivable that for other models the fully symbolic approach is the best-suited one.

6.5.1 Conclusion

In this chapter, we have shown how game-based abstraction for PA can be leveraged to obtain lower and upper bounds for both minimal *and* maximal reachability probabilities. The latter sets it apart from a naive abstraction in terms of PA. This is achieved through separating the nondeterminism inherent to the model from the nondeterminism that is introduced by the abstraction. As the abstraction process leverages predicate abstraction and extracts the abstract games directly from a symbolic description given as either **PRISM** or **JANI**, the technique is able to handle PA with infinite state spaces. It is, to the best of our knowledge, the only automated technique to compute minimal and maximal unbounded reachability probabilities for *general infinite* PA.

So, when is game-based abstraction refinement useful in practice? The most obvious case is the application to infinite PA. For such models, it provides the only push-button technique to obtain lower and upper bounds on reachability probabilities. An evaluation of our prototypical implementation in the framework of Storm shows that game-based abstraction is able to prove tight bounds on small abstractions for infinite models. However, our experiments show that also for finite state benchmark models, the abstraction can result in significant savings in both space and time. In particular, our prototype is competitive with Pass [Hah+10b], the only other publicly available implementation of game-based abstraction for infinite-state **PRISM** programs, which is no longer maintained. Leveraging the infrastructure of Storm, our prototype goes beyond Pass in that it can compute *sound* lower and upper bounds *using rational arithmetic*.

Clearly, the abstraction process is more effective if large parts of the state space can be summarized into few blocks without causing a large imprecision. Also, our experiments show that typically non-trivial bounds can be derived quickly, whereas approaching the actual result very closely may take substantially longer and require a much finer abstraction. This suggests that the technique may, for instance, be applied fruitfully in a setting in which the emphasis is on a rough approximation and the precision of the result is of less importance.

There are several interesting directions for future research. Since the abstraction process typically dominates the runtime, further optimizations could result in significant speedups. This could, for example, be an *on-the-fly* abstraction process that does not enumerate all solutions but builds the transitions in an on-demand manner like the one implemented in Pass. Also, our evaluation shows that the quality and quantity of the predicates is of utmost importance for the number of refinements and runtime. It therefore appears worthwhile to *spend more computational effort on trying to derive predicates* that are more likely to improve the overall progress rather than heuristically picking a single point of refinement. Finally, an extension towards a *richer class of properties* would clearly be desirable. In particular, reachability rewards can be computed similarly to reachability probabilities on PA and could therefore be amenable to menu-game-based abstraction. This would cover many interesting properties from the domains of performance analysis and artificial intelligence.

Chapter 7

Storm - A Modern Probabilistic Model Checker

7.1 Motivation and Goals

Prism is the most well-known probabilistic model checker. It combines sophisticated techniques for a range of probabilistic models and properties. It has been downloaded by more than sixty thousand users to date and its fourth tool paper [KNP11] alone is cited by more than 1300 other scientific papers. Numerous ideas and algorithms have been developed and integrated into Prism over the past two decades. Because of this impressive success story, the main developers of Prism, Dave Parker (University of Birmingham), Gethin Norman (University of Glasgow) and Marta Kwiatkowska (University of Oxford) were awarded the prestigious HVC award in 2016. The researchers were honored for "the invention, development and maintenance of the PRISM probabilistic model checker" and in particular for "their outstanding contributions to probabilistic model checking and, more generally, to formal verification".

Despite the domination of Prism in the field and despite other existing probabilistic model checkers such as Mrmc [Kat+11], in the beginning of 2012 we decided to develop a new model checker called Storm[1].

To better motivate our decision, let us briefly recapitulate the development goals of Storm. First and foremost, we wanted to have an easy-to-use platform for experimenting with new verification algorithms, richer probabilistic models, algorithmic improvements

[1]`http://www.stormchecker.org/`

and various new features. As this typically involves extensive experimentation, it requires abstraction and identification of building blocks that can be easily substituted for one another at key positions within the tool. In particular, easy interfacing with other libraries allows for reusing highly optimized code as well as integrating new functionality. Simultaneously, modern techniques more and more move from simple model checking queries to performing model checking "in-the-loop". This creates the often competing goal to consistently value high performance and not introduce potential bottlenecks. Based on the observation that to date there is no "one-size-fits-all" solution for the analysis of probabilistic systems, a modern probabilistic model checker needs to offer a variety of engines. For example, symbolic methods (based on decision diagrams or SMT solvers) have the potential to scale to systems with huge or even infinite state spaces, but not all models allow for a compact symbolic representation. Conversely, explicit state model checking typically offers high performance, but suffers from the inherent memory barrier.

It cannot be denied that these goals could have been potentially achieved by extending Prism or Mrmc in a suitable manner. Let us start with our motivation not to continue Mrmc. From the very beginning Mrmc pursued the goal of being a maximum performance backend. Written in highly optimized **C**, it was able to outperform Prism in many cases where the model is not too large [Zap08]. Given its main focus, it does not provide a symbolic input language but rather requires an explicit input format that lists all transitions and their probabilities. Consequently, Mrmc only provides explicit state probabilistic model checking. That is, the systems are always internally represented by explicit (sparse) matrices. As argued in the previous chapters, symbolic languages and symbolic methods allow for much greater scalability for suitable models and Mrmc is not built to be easily extendible in this direction.

Let us turn to Prism. When we started developing Storm, Prism did not have support for fully explicit state model checking. Instead, all its engines relied on multi-terminal binary decision diagrams (MTBDDs) and were therefore somewhat susceptible to input models that cannot be compactly represented in these data structures. We also wanted to use the model checker as a testbed for other research activities, which entailed that the tool had to be flexible enough to exchange solvers and model checkers for one another and also that it can easily interface with other tools and libraries to benefit from new techniques and existing high-performance implementations. For example, more and more modern (symbolic) techniques apply satisfiability modulo theories (SMT) solvers, whose performance increased substantially over the past decade. However, for technical reasons, Prism does not interface with an Smt solver in its current release. Unofficial extensions to Prism occasionally include support for Smt, but are either limited to **Java**-based solvers or have to make use of the comparatively slow and technically cumbersome **Java** native interface (JNI). Similarly, Prism implements the numerical solution algorithms itself and does not benefit from advances in the field of numerical algorithms. Again, we want to stress that it would have been possible to extend Prism in this direction. However, we set out to develop a new probabilistic model checker Storm.

7.2 Competitors

In the following, we compare Storm with its main competitors on the level of features as well as performance. We therefore now first briefly summarize the most important aspects for the other tools. Note that this summary is incomplete and — considering further progress of the tools — necessarily only a snapshot.

Prism. Arguably the most famous probabilistic model checker, Prism[2] [KNP11] is developed almost since two decades. It supports discrete- and continuous-time Markov chain, PA and PTA (and multiplayer stochastic games via an extension called Prism-games). For these models, it supports the verification of a wide range of properties such as CTL, PLTL, PCTL and continuous stochastic logic (CSL) as well as expressive extensions in the direction of expected rewards, conditional probabilities and rewards [Bai+14; Mär+17; Bai+17a] and quantiles [Kle+18]. To handle a large range of models, Prism offers four separate engines. The recent explicit engine builds and verifies the model in an explicit state representation. All other engines (the (semi-)symbolic engines) first build an MTBDD representation of the system and then differ in the way the numerical solution is conducted. While the mtbdd engine carries out all computations on MTBDDs, the sparse engine translates the transition MTBDD into a sparse matrix format and then solves the numerical queries on the latter. Finally, the hybrid (the default) engine builds a cross-over between these two and tries to approach the memory requirements of the mtbdd engine and preserve the performance of the sparse engine. Prism features a dedicated graphical-user interface that improves the usability with in a variety of ways. For instance, models can be simulated and the analysis results can be plotted as graphs. Prism is written in **Java**, but the old engines invoke code written in **C** at performance-critical spots and to interface with the DD library CUDD.

Epmc. Epmc[3] (short for *extendible probabilistic model checker*) is mostly developed at the Institute of Software Chinese Academy of Sciences (ISCAS). It extends its predecessor IscasMC [Hah+14] in several directions. For instance, the latter only supported PLTL

[2] `http://www.prismmodelchecker.org/`

[3] available at `https://github.com/liyi-david/ePMC`

and EPMC extends this to PCTL*, CSL, expected rewards and transient properties. Regarding model types, it can treat discrete- and continuous-time MCs, PA, multiplayer stochastic games and probabilistic parity games (PPGs). It features two distinct engines, namely the explicit engine that builds and verifies the model only using an explicit representation of the system and the dd engine that uses DDs for both steps. A static website serves as the graphical-user interface of EPMC and allows to graphically perform tasks such as model and property management. Just as PRISM, EPMC is written in **Java** but also uses natively compiled code, for instance to integrate DD libraries (by default CUDD) written in **C**.

MODEST TOOLSET. The MODEST TOOLSET[4] [HH14] is a collection of tools dedicated to the analysis of networks of stochastic timed automata (SHA), a very rich formalism that allows for (i) nondeterministic choices, (ii) continuous system dynamics, (iii) stochastic choices, (iv) nondeterministic delays, and (v) stochastic delays. In particular, SHA subsume PTA, PA, MA and therefore, by extension, also discrete- and continuous-time MCs. As the name suggests, the MODEST TOOLSET consists of several tools. Among them is a statistical model checker called MODES, a safety model checker for SHA called PROHVER and, finally, a model checker MCSTA that can treat a more limited set of models such as PTA and PA. In the performance evaluation, we focus entirely on MCSTA since it is the most comparable to the other mentioned tools. MCSTA is an explicit state model checker and does not employ symbolic data structures. Instead, it has three modes that can be considered engines. The memory engine keeps all parts of the model and all vectors needed in the solution process in memory. In contrast, disk writes the transition matrix and solution vectors to disk and, if memory pressure becomes high, removes infrequently used parts to disk. Finally, hybrid is similar to disk but keeps more information in memory during the model building step. A remarkable property of MCSTA in comparison to the other mentioned tools is its model building technique that makes use of just-in-time compilation. In a nutshell, at runtime source code is generated and emitted that is then executed and thereby explores the model. This can speed up model building significantly. The MODEST TOOLSET is written in **C#** and unlike PRISM, EPMC and STORM, its source code is not publicly available.

7.3 Technicalities

Before we move on to a present the features and the higher-level design choices of STORM, we want to mention a few technical details that partially affect the former. By far the largest part (around 140,000 lines of code) of STORM is written in the **C++**

[4]`http://www.modestchecker.net/`

programming language and extensively uses template-based meta-programming. This has several positive and negative implications. On the one hand, it serves the purpose of high performance for several reasons. First, **C++** allows fine-grained control over implementation details like memory allocations. Secondly, **C++** templates allow code to be heavily reused while maintaining performance as the static polymorphism enables type-dependent optimizations at compile-time. Finally, we observe that many high-performance solvers and data structure libraries that are well-suited for the context of (probabilistic) verification are written in **C** or **C++** (and also partially make use of template meta-programming), such as

- most SMT solvers (Z3 [MB08], MathSat [Cim+13], Cvc [Det+14], Yices [Dut14], Smt-Rat [Cor+15]),
- LP solvers (Gurobi [Gur16], glpk[5]),
- linear algebra libraries (Gmm++[6], Eigen[7] [G+10]),
- DD libraries (CUDD [Som], Sylvan [Dij16]), and
- rational numbers and function libraries (CArL [Cor+15], GMP[8], GiNaC [BFK02], CoCoALib [AB17]).

Choosing **C++** as the language for Storm therefore allows easy and fast interfacing with these solvers. On the other hand, the advantages come at a price. Advanced templating patterns can be difficult to understand and increase compile-times significantly. Currently, building all binaries related to Storm from source code takes about half an hour on a modern laptop. In comparison, building Prism takes under 2 minutes even though it contains more lines of code.

7.4 Features

In this section, we will take a glance at the features and limitations of Storm. We will highlight how Storm differs from its main competitors. First and foremost, we want to mention that the feature sets of the model checkers are *incomparable* in the sense that each of them can solve some verification-related tasks that the other cannot do.

[5] `https://www.gnu.org/software/glpk/`
[6] `http://getfem.org/gmm.html`
[7] `http://eigen.tuxfamily.org/`
[8] `https://gmplib.org/`

This is simply a result of the different focus of the tools. For example, Storm does currently not support PLTL model checking while Prism has support for it and Epmc even employs sophisticated multi-layered lazy constructions to increase the scalability of PLTL verification further [Hah+15]. Similarly, Storm neither offers surrogates for all of Prism's verification engines (see Section 7.4.5) nor provides verification based on discrete-event simulation (which is also known as statistical model checking) whereas Modest and Prism do. Finally, Modest, for instance, goes beyond the other tools by providing support for models with more general distributions and continuous behaviors governed by differential equations.

In the following, we try to give an overview about what makes Storm's characteristics unique. For these, let us use the abstract control flow diagram in Figure 7.1 to structure our enumeration. We want to remark that in the following — just as for the other tools — we abstract from certain details and that not all combinations of features are available.

7.4.1 Model Types

Storm supports the analysis of Markov chains (MCs) and probabilistic automata (PA). More specifically, it can deal with discrete-time and continuous-time versions of these formalisms. In total, this yields four different model types: classical discrete-time Markov chains and continuous-time Markov chains, as well as probabilistic automata and Markov automata. A categorization of these models and more details can be found in Section 2.2.

Clearly, MA are the richest model in the sense that PA (and by extension also DTMCs) can be seen as MA without (exponentially distributed) delays, and CTMCs can be seen as MA without nondeterminism. All these model types can be enriched with state and transition-based reward (or dually: cost) structures. While Epmc, Prism and Storm all support MCs and PA, only few other model checkers, such as IMCA [Guc+12] enable the analysis of (timed properties on) MA.

7.4.2 Modeling Languages

Most probabilistic model checkers support one or two modeling languages. More specifically,

» Prism supports the **PRISM** language as well as an input format that explicitly enumerates the transitions of the system similar to that of Mrmc,

» Epmc supports the **PRISM** language as well as **JANI** (see Chapter 3), and

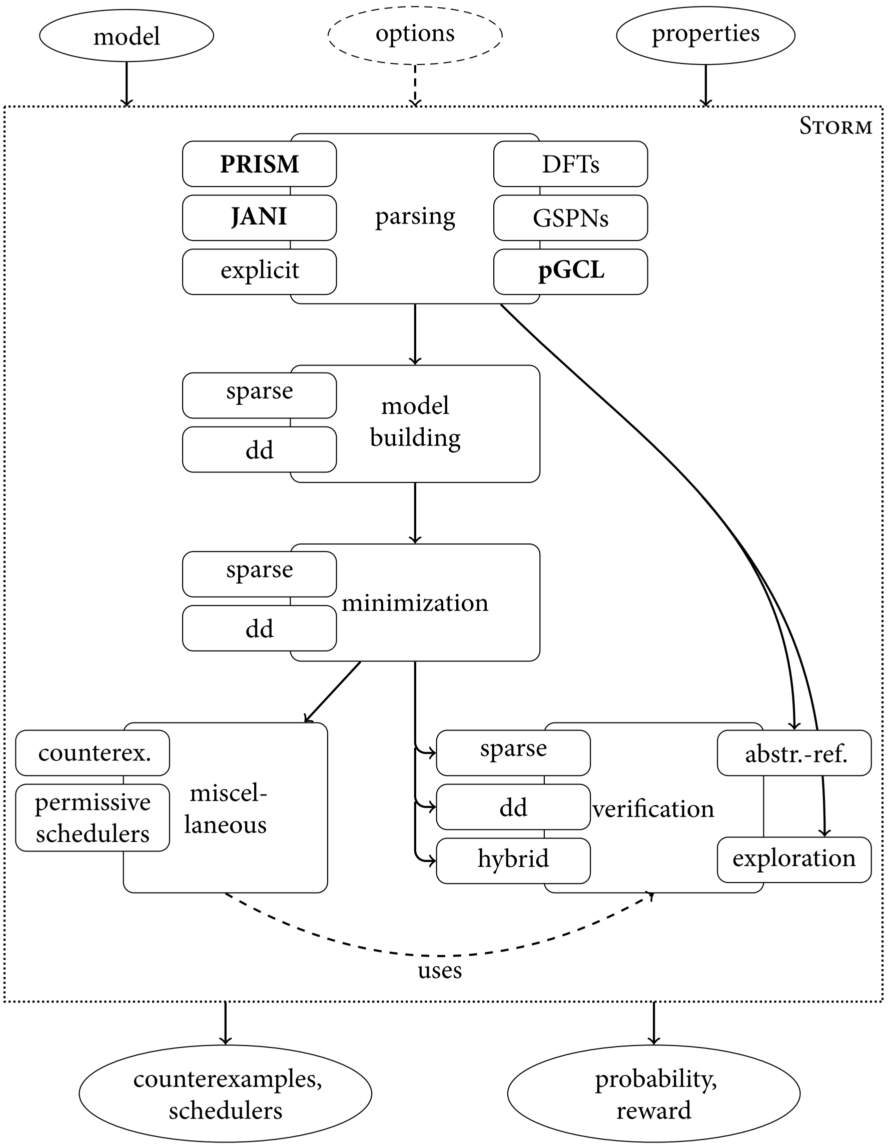


Figure 7.1: The abstract control-flow within Storm.

» MODEST supports the **Modest** language as well as **JANI**.

With the exception of the **Modest** language, all of these languages are also supported by STORM. However, it supports three other modeling languages. First, the user can input a generalized stochastic Petri net (GSPN) [MCB84], which is then translated to **JANI** automatically. Secondly, dynamic fault trees (DFTs) are a means to specify the fault behavior of systems that is widely used in industry [RS15]. Due to dedicated state space generation and reduction techniques, STORM has been shown to significantly outperform competing tools in this domain [VJK16]. Finally, a recent trend in the analysis of probabilistic systems is probabilistic programming. The latter refers to programs written in a probabilistic extension of regular sequential programs, such as **pGCL** [HSM97], that can additionally be extended with statements expressing conditional reasoning, an ingredient that is essential to describe Bayesian networks. STORM can parse and translate programs written in **pGCL** to **JANI**, which makes such programs amenable to existing probabilistic model checking techniques such as standard value iteration (for finite state programs) or game-based abstraction refinement (for infinite state programs).

7.4.3 Options

For completeness reasons, we have also included "options" as an input to STORM. In our view, it is fair to state that all mentioned tools are highly customizable through the options they provide. While this may be advantageous if the user can encode helpful domain knowledge, the number of options of all tools may be overwhelming to users that are not familiar with the inner workings of the tools. As of now, the (main) binaries of STORM, PRISM and EPMC each offer more than ninety options. While all tools provide reasonable defaults, they cannot be adequate for all inputs from a very wide range of input models. We therefore see an interesting direction for future research in the analysis of the input with the goal of deriving a set of suitable options for the specific instance, an approach that was already suggested in the non-probabilistic setting [Dem+17].

7.4.4 Properties

As we already mentioned, STORM lacks support for PLTL. Instead, at its core, it focuses on branching-time logics such as PCTL for discrete-time and CSL for continuous-time models. In particular, these logics subsume the most important properties in the form of reachability queries. For example, similar to the non-probabilistic setting, model checking ω-regular properties on probabilistic systems reduces to a reachability problem on a suitable product construction where the main difference to the traditional setting is that the automaton needs to satisfy certain criteria, e. g. be (limit) deterministic or unambiguous. The properties accepted by Storm can also reason about the rewards or costs in the system. In this regard, the tool mainly focuses on expected rewards:

» what is the expected reward that is accumulated until a set of goal states is reached?

» what is the expected reward that is accumulated until time point *t*?

» what is the instantaneous reward received at exactly time point *t*?

A class of properties that has received further attention in the past years [Bai+14; Bai+17a] are conditional probability and reward queries, respectively. Storm can, for instance, compute the probability that a certain set of states is reached *provided* that some other state set is visited, too, and similarly for rewards. Moreover, Storm was recently extended with support for multi-objective queries [QJK17]. Here, the goal is to find good trade-offs between possibly conflicting objectives in models that involve nondeterministic choices. Similarly, cost-bounded properties [Har+18] are a recent addition and allow determining the probability to reach a set of target states within a certain cost bound.

7.4.5 Engines

We have already mentioned that in probabilistic model checking and probably in verification in general, there is no "one-size-fits-all" solution. Instead, it heavily depends on the input model and the properties, which techniques and tools perform best. Therefore, current model checkers opt to offer a range of engines that drive the model building and verification steps. In this regard, Storm is not different. More specifically, it features two different in-memory representations of probabilistic systems. First, it can use sparse matrices, an explicit representation form that uses memory roughly proportional to the number of transitions with non-zero probability. Sparse matrices are suited for small and moderately-sized models and allow for fast operations also on models with irregular structure. Secondly, it can store models symbolically using MTBDDs, which are (sometimes) able to represent gigantic models very compactly. This typically comes at the price of more expensive operations, in particular for the numerical solution. Most of Storm's engines are built around these model representations. The sparse engine exclusively uses the sparse matrix-based representation. It first builds the matrix representation of the reachable state space and then analyzes the model using the standard (numerical) approaches. While the exploration engine also uses sparse matrices, it uses ideas from machine learning to avoid building the full system [Brá+14]. Instead, it proceeds in an "on-the-fly" manner and explores those parts of the system that appear to be most relevant to the verification task. The other engines use the MTBDDs as their primary form of representation. Except for the concrete in-memory representation, the dd engine is the counterpart to the sparse engine in the sense that model building and verification is done on the very same representation and no translation takes place. STORM's hybrid engine tries to avoid the costly numerical operations on MTBDDs by transforming only the relevant parts of the system into a sparse matrix representation. By this, it follows the approach of PRISM's sparse engine and is not to be confused with the latter's hybrid engine, which is to be classified as "more symbolical". Finally, the abstraction-refinement engine implements the technique described in Chapter 6 and is able to compute bounds for both minimal and maximal reachability probabilities for infinite PA (and probabilistic programs).

7.4.6 MTBDDs

To realize the support for DDs-based representations of systems, STORM relies on two different libraries: CUDD [Som] and SYLVAN [Dij16]. While the former is very well established in the field, the latter is more recent and tries to make use of modern multi-core CPU architectures by parallelizing costly operations. This comes at the price of more expensive bookkeeping and in general CUDD performs better if there are many operations on smaller DDs and SYLVAN is faster when fewer operations on larger DDs are involved. Given that modern CPUs tend to have more and more cores, SYLVAN offers an interesting trade-off by trying to compensate for the slower operations on DDs (compared to explicit representations) by making efficient use of these resources. We want to remark that STORM implements an abstraction layer on top of the two libraries that uses static polymorphism. This way, it is possible to write code that is independent of the underlying library and does not incur runtime costs. This also means that STORM can easily be extended with more DD libraries as they become available. This sets it apart from PRISM, which, because of its long-standing history, is somewhat coupled with CUDD.

7.4.7 Bisimulation Minimization

As we have shown in Chapter 5, bisimulation minimization is able to reduce model sizes drastically while preserving all properties of interest. STORM implements this reduction using both the symbolic approach presented in Chapter 5 as well as its counterpart operating on sparse matrices. This allows applying the reduction on a wider range of models: if the model is too large to be treated in an explicit representation, the explicit bisimulation minimization cannot treat the model independent of the degree of symmetry in the model. For large, structured models, this is overcome by building and minimizing the model in an MTBDD-based representation.

7.4.8 Exact Arithmetic

Several works [Wim+08; Wim10; Bau+17] observed that the numerical methods applied by probabilistic model checkers are prone to numerical errors. This has mostly two reasons. First, the floating point data types used by the tools are inherently imprecise. For example, representing the probability $\frac{1}{10}$ using IEEE 754 compliant double precision introduces an error of $5 \cdot 10^{-18}$. In the presence of numerical algorithms, these errors accumulate and may lead to incorrect results. Secondly, the numerical algorithms sometimes themselves are strictly speaking unsound. For example, value iteration (see Chapter 6) approximates the solution in the limit, but the termination criterion implemented by most tools does not guarantee that the obtained result is differing by at most the given precision ϵ from the actual solution. One way to combat this is to approach the solution from both directions, a technique referred to as *interval iteration* [HM14]. STORM implements the latter and additionally an even more recent algorithm called *sound value iteration* [QK18]. We want to remark that using policy iteration [How64] is also problematic in the case of numerical inaccuracies of the underlying linear equation solver.

An alternative to the above is to employ *rational arithmetic*. That is, probabilities and rewards in the model are stored as rational numbers. Then, systems of linear equations may be solved without introducing any errors and techniques such a policy iteration allow for solving Bellman equations without error. STORM implements these ideas and allows for the exact solution of many properties. Through the use of **C++** template meta-programming, large parts of the code are written agnostic of the data type (floating point, rational number or even rational functions) and only the core parts are specialized based on the data type. As this happens at compile-time, no runtime cost is incurred. More specifically, two techniques are offered to solve systems of linear equations using rational arithmetic. The first is based on Gaussian elimination and the second on a recent technique called *rational search* [Bau+17]. The idea of the latter is to use an imprecise solver to approximate the exact solution and then sharpen this to a precise rational solution using the Kwek-Mehlhorn algorithm [KM03]. If a straightforward check then returns that the sharpened values constitute an actual solution, the technique can return it. Otherwise, the precision of the imprecise underlying solver is increased and the loop is restarted.

7.4.9 Parametric Models

STORM features the analysis of models that replace one or more probabilities or rewards in the model with *parameters*. They may appear at multiple positions in the model and have a fixed but unknown value. Typically, valid ranges of all used parameters are assumed in the form of intervals. Then, for example, the probability to reach a certain set of states in a parametric DTMC is no longer a value, but rather a rational function [Daw04]. This function returns the probability based on a concrete input parameter valuation. There are many interesting questions that one can ask revolving around parametric systems. One of them is *parameter synthesis* where the goal is to decompose the parameter space into regions in which a predefined property is either satisfied or violated. Such a decomposition indicates *all* admissible parameter values among which the system engineer might choose, possibly according to additional criteria. Storm is used as a backend of the parameter synthesis tool PROPhESY [Deh+15] that provides a web-based user interface to visualize the behavior of the system over the parameter space. It provides two modes to perform parameter synthesis. One is based on computing the aforementioned rational function through state elimination [Daw04] that can also be seen as Gaussian elimination. The other avoids computing a potentially large rational function and determines validity of a formula over a region of parameter valuations through a reduction to a non-parametric system and is referred to as the *parameter lifting* approach [Qua+16]. In contrast to other approaches, the latter extends naturally to parametric PA and for both techniques, Storm has proven to outperform other state-of-the-art tools such as Prism or Param [Hah+10a]. In the case of computing rational functions, Storm follows the same principle as the other tools, but obtains speedups of up to two orders of magnitude through an optimized implementation as well as the use of rational function representation provided by the library CArL [9]. It uses a partially factorized representation of rational functions that allows for faster cancellation of terms in the numerator and denominator of the function through cheaper computations of the greatest common divisor. Recently, Epmc was extended with a technique to compute rational functions for the purpose of sampling the model [GHS18] where it is less important to cancel rational functions as far as possible.

7.4.10 Counterexample generation and Permissive Scheduler Synthesis

In the past decade, the feature portfolios of probabilistic model checkers have increased significantly. Initially, the focus lay mostly on computing the probability that a certain event happens. However, as witnessed by parameter synthesis there are many other valuable results a probabilistic model checker can produce. Among them is also the synthesis of counterexamples as a feedback for the developer of the system under analysis. Counterexamples are essential to the acceptance of formal methods among engineers and help drive the design cycle. Storm supports the generation of high-level counterexamples for both Prism and **JANI** input models as presented in Chapter 4.

[9]available at `https://github.com/smtrat/carl`

In a similar spirit, scheduler synthesis aims at constructing a strategy to resolve the nondeterminism that is optimal with respect to a given objective. This can be leveraged to solve (stochastic) planning problems as they, for instance, arise in robotics. Here, the choices are modeled as nondeterminism and an optimal scheduler in a stochastic model enriched with costs corresponds to a plan that is optimal with respect to the incurred costs. As robots need to be able to adapt to unknown terrain, it may be the case that the cost function is not known a-priori but is learned by the robot as it explores its surroundings. However, the robot must not perform any actions that risk its safety, such as moving to a dangerous position. This effectively means that the synthesized optimal scheduler needs to respect a set of safety constraints resulting in a multi-objective problem. Storm implements the synthesis of *permissive schedulers* [Drä+15] that satisfy a set of safety constraints but try to remain as permissive as possible in the sense that as little behavior as possible is ruled out. More details on the two supported techniques (one Milp-based and one Smt-based) can be found in [Jun+16].

7.4.11 Application programming interfaces (APIs)

There are three ways to access the features of Storm. Just like all mentioned tools, it offers a command-line interface in the form of a set of binaries. They group the features and settings for better usability. For example, the functionality related to DFTs or the verification of parametric models are placed in separate executables. In contrast to Prism, Storm does not have a graphical user interface.

One of the two real APIs Storm offers to developers is **C++**-based. It allows fine-grained and performance-oriented access to all of its features. However, because of the involved templating mechanisms, this API is clearly dedicated to advanced developers. If performance is a subordinate goal and the ease of use is a primary concern, a better choice is to use the **Python** API dubbed Stormpy[10]. The latter abstracts from many technicalities and therefore offers a much better user experience. This allows for rapid prototyping while benefiting from the encapsulated high-performance **C++** implementations.

7.5 Solvers

Probably the most outstanding trait of Storm's architecture is the concept of *solvers*. Ultimately, many tasks related to (probabilistic) verification revolve around solving subproblems. For example, computing reachability probabilities or expected rewards in a DTMC reduces to solving a system of linear equations. This is similar for PA in

[10]available at `https://github.com/moves-rwth/stormpy`

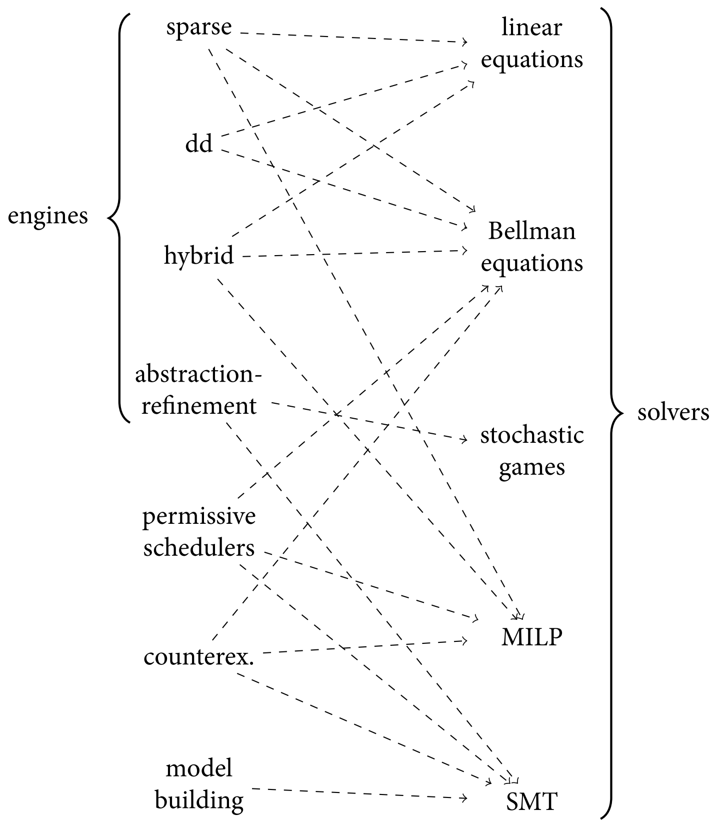


Figure 7.2: Solvers used by Storm.

that a system of equations needs to be solved with the difference that the equations are Bellman equations and involve minima and maxima. However, these are by no means the only kinds of problems appearing in probabilistic verification. Figure 7.2 illustrates which functionalities of Storm have a dependency to one or more solvers. For example, (explicit) model building employs Smt solving. As the initial states of symbolic models (e. g. Prism or **JANI**) are given by the satisfying assignments of an expression, Storm uses Smt solvers to enumerate the possible initial states. Similarly, the extraction of the abstract model from the symbolic model (as presented in Chapter 6) in the abstraction refinement engine crucially depends on enumerating satisfying assignments and therefore Smt solvers. As yet another example, consider the synthesis of high-level counterexamples as in Chapter 4. Here, one of the offered techniques relies on the solution of a MILP while the other uses SMT solvers.

Recall that two of the main goals in the development of Storm were the ability to exchange central building blocks (like solvers) and to benefit from (re)using high-performance implementations provided by other libraries. It therefore offers abstract interfaces for the solver types mentioned above that are oblivious to the underlying implementation. Offering these interfaces has several key advantages. First, it provides easy and coherent access to the tasks commonly involved in probabilistic model checking. Secondly, it enables the use of dedicated state-of-the-art high-performance libraries for the task at hand. More specifically, as the performance characteristics of different backend solvers can vary drastically for the same input, this permits choosing the best solver for a given task. Licensing problems are avoided, because implementations can be easily enabled and disabled, depending on whether or not the particular license fits the requirements. Finally, implementing new solver functionality is easy and can be done without detailed knowledge of the global code base. This allows to embed new state-of-the-art solvers in the future. For each of the solver interfaces, several actual implementations exist. For example, Storm currently has four implementations (each of which with a range of further options) of the linear equation solver interface for problems given as sparse matrices: one is based on Gmm++, one on Eigen, one uses its internal data structures and algorithms for numerical algorithms and another one is based on Gaussian elimination [Daw04]. Table 7.1 gives an overview over the currently available implementations. Here, all solvers that are purely implemented in terms of Storm's data structures and do not use libraries are marked with an asterisk to indicate that they are "built-in".

7.6 Evaluation

As previously mentioned, all tools have their strengths and weaknesses regarding the set of features they offer. In this section, we want to compare how the tools Prism, Epmc, Modest (more specifically: mcsta) and Storm perform on standard probabilistic model checking tasks, such as computing reachability probabilities or expected rewards.

Before we move on, we want to say some words about the meaning of the following figures. We already pointed out that the tools come with a large number of options, some of which have a continuous domain. For the evaluation, we only modify key settings like the selected engines and leave the rest as they are by default. Self-evidently, this need not be the optimal settings for the tools for the selected benchmark models and it may

solver type	available solvers
linear equations (sparse)	Eigen, Gmm++, Gaussian elimination*, native*
linear equations (MTBDD)	CUDD, Sylvan
Bellman equations (sparse)	Eigen, Gmm++, native*
Bellman equations (MTBDD)	CUDD, Sylvan
stochastic games (sparse)	native*
stochastic games (MTBDD)	CUDD, Sylvan
(MI)LP	Gurobi, glpk
SMT	Z3, MathSat, **SMT-LIB** [BFT15]

Table 7.1: The solvers Storm provides out-of-the-box.

well be the case that performance varies drastically with a different set of options and or benchmarks. After all, seemingly minor options may have a huge effect on performance. For instance, when performing state elimination for parametric systems, the runtime may vary between few seconds and hours depending on the selected heuristic for ordering the states [Deh+15].

Furthermore, the selection of models plays a crucial role. To remain as unbiased as possible, we draw all models from Prism's benchmark suite [KNP12]. In particular, we take *all* 7 DTMCs, 7 CTMCs and 9 PA models from this suite (23 in total). Most of these models are scalable in one or more parameters. For each of the models, we therefore consider several parameter instances. Furthermore, every model comes with a set of properties and we consider at least one instance of each model for each property associated with the model (13 properties on DTMCs, 37 on CTMCs and 31 on PA). Overall, the properties cover unbounded and bounded reachability probabilities, expected rewards (reachability, cumulative and instantaneous) as well as long-run probabilities and rewards. In total, we have 330 verification tasks: 60 on DTMCs, 149 on CTMCs and 121 on PA. More details can be found in Appendix F.

A little less obvious is the effect of the operating system and hardware on the results. The appendix of the extended version of [Mär+17] [11] impressively showcases how runtimes are affected by the concrete system the experiments are run on. Here, runtimes vary as much as 300% between systems even though the underlying hardware is roughly comparable. The authors conclude that comparisons need to be taken with a grain of

[11]available at `https://wwwtcs.inf.tu-dresden.de/ALGI/PUB/SEFM17`

salt and only carry over to other systems on a coarse level. Even on the same system, the results can vary over separate runs of the tools and it would in principle be advised to average over a large number of runs. However, we decided to rather have a very broad comparison on a large number of models and due to the sheer number of experiments (we run several configurations of the tools on every verification task), it was not practical to repeat individual experiments sufficiently often to rule out noise in the data. In any case, we believe that for instances that are not solved within a few seconds, the noise will overall be negligible.

Finally, we compare the tools mostly in terms of runtime. While this is an interesting metric, it is certainly not the only one. As probabilistic model checking tends to not only be time-consuming but also memory-intensive, another comparison metric would be the memory requirements of the tools. This is somewhat difficult since Prism, Epmc and Modest are mainly written in garbage-collected languages. In practice, the virtual machines (in particular the **Java** VMs) running the executables quickly use all available memory and only free memory as necessary. The peak memory usage is therefore not representative of the memory that would have been necessary to successfully complete the verification task. Similarly, for all DD-based tasks, the sizes reserved for the libraries (CUDD and Sylvan) need to be given a-priori (for all the tools) and peak memory usage does not reflect the actual requirements. We therefore resort to setting an identical time and memory limit for all tools and determine which of the tools was able to complete the task within the resource limit. If a task could not be completed because of missing resources, we indicate whether it ran out of memory (MO) or it timed out (TO).

We hope that the evaluation provides a rough idea of the strengths and weaknesses of the tools, but ultimately *it depends on the concrete model, property, OS and hardware which of the tools performs best.*

7.6.1 Technical Setup

For the experiments, we use an HP BL685C G7 blade with 192GB of memory and 48 cores (each clocked at 2.0GHz) running 64-bit Debian 9 ("Stretch"). Every experiment was restricted to 16GB of memory and one hour. Both **Java**-based tools (Prism v4.4dev [12] and the most recent Epmc revision[13]) were compiled and run using Oracle's **Java** development kit[14] v10.0.2 whereas mcsta (v3.0.104-g6e5bc65) was run under Mono[15] v5.4.1.6. With the exception of the parallel garbage collection of the **Java** virtual machine,

[12] commit dc5a8b6f7e2b02093d986a52ee344bc183e40cd0 in Prism's github repository

[13] commit a6b2c87e034a20f5775e7ca62f67eed52d325994 in Epmc's github repository

[14] available at `http://www.oracle.com/technetwork/java/javase/downloads/`

[15] available at https://www.mono-project.com/

which we limited to 4 threads, all experiments were limited to one thread. As we already mentioned, we set a minimum number of options for the tools and otherwise rely on the defaults. Among the specified options is the (relative) precision $\epsilon = 10^{-6}$ since the tools' default behaviors regarding precision differ significantly and the results can then not be compared in a meaningful way. While it is known that the (default) techniques used by all tools are not sound in the sense that they guarantee to be at most ϵ away from the actual solution, some of the results were too imprecise. We therefore mark all incorrect results as "error" where incorrect is defined as not achieving a (relative) precision of 10^{-2} with respect to the reference result. For a given verification task the latter is determined as follows:

» if at least one of the model checkers (of the ones that support it, i. e. Storm, Prism) was able to compute an exact result in rational arithmetic, use it as the reference result (and also compare it to the other exact results that could be obtained),

» if no exact result could be obtained, but at least one of the model checkers (of the ones that support it, i. e. Storm, Prism, mcsta) was able to compute a result using a *sound* method ([Bai+17b; QK18]), use it as the reference result (and also compare it to the other sound results that could be obtained),

» if none of the above was successful, determine for each model checker with how many of the other model checkers it agrees and finally allow all (potentially different) results of the model checkers with the highest scores as reference results.

We do realize that it is not strictly guaranteed that this procedure arrives at a reference result that is within ϵ-distance of the actual result, but we argue that it is the best approximation with the data we have.

7.6.2 Methodology

In our comparison, we first compare only those engines of the tools that are similar in spirit and only then compare the performances of the tools when selecting the *best* engine for *each* instance. We focus on runtimes and measure the wall-clock runtimes (including model building and model checking) for all experiments. Note that each experiment is run in isolation and no information (such as state spaces or previous results) is reused by any of the tools between different experiments. For each of the comparisons, we provide two classes of graphs that visualize the experimental results. First, we compare the performance of Storm with each of the other tools one-by-one separately and give the results in the form of *scatter plots*. The latter indicate the type

(DTMC, CTMC or PA) of the verification task by means of different marks. The scatter plots use logarithmic scales on both axes and indicate speedups of 10 and 100 by means of dotted lines. Moreover, points for experiments that ran out of resources (time-out or memory-out), resulted in an error or are not supported are drawn on separate lines. Here, "error" does not only refer to incorrect results but may also refer to any other problem that prevented the tools from returning an answer. Secondly, we measure the overall performances of the (comparable) engines of the tools and draw them in *quantile plots* similar to those used by popular competitions, for example the Competition on Software Verification (SVComp). Such a graph expresses how many benchmark instances (measured on the x-axis) *each* were solved in at most the time given on the y-axis. In other words, the point $\langle x, y\rangle$ is contained in the quantile plot for tool Z if the *maximal* runtime of Z on the x fastest instances (for Z) that could be solved is y seconds. In particular, this means that times are *not* cumulative and do therefore *not count the start-up times of tools repeatedly*. To not over-emphasize very low runtimes (e. g. resulting from different start-up), we adopt the strategy of SVComp and use a linear scale (on the y-axis) for the points representing runtimes up to one second and a logarithmic scale (also on the y-axis) for the runtimes higher than one second. Note that time- and memory-outs, errors and unsupported experiments may skew the lines of the affected tools as all these outcomes do *not* count as solved.

7.6.3 Explicit-State Probabilistic Model Checking

We start with a comparison of those engines of the tools that build an explicit representation of the system (typically in terms of a matrix) and perform verification using this representation, an approach that is typically referred to as *explicit state model checking*. More concretely, these are

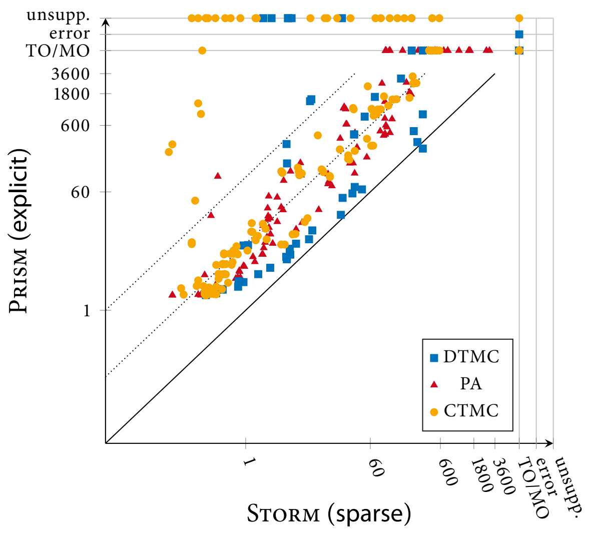
» Prism's explicit engine,

» Epmc's explicit engine,

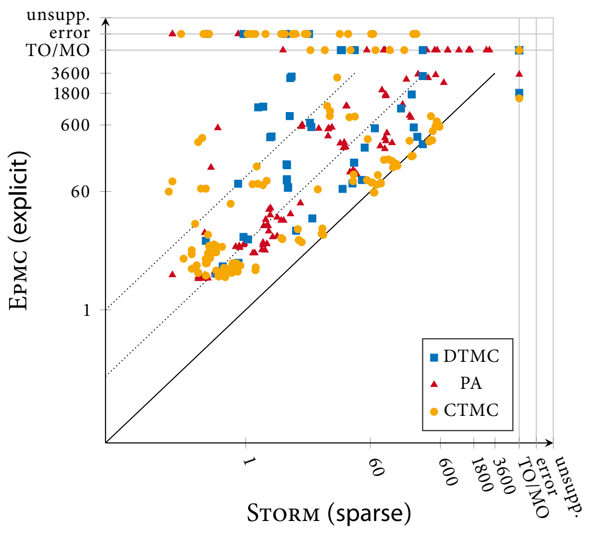
» all of mcsta's engines (memory, hybrid, disk), and

» Storm's sparse engine.

Figures 7.3(a) and 7.3(b) show the scatter plots for the comparison of Storm's explicit engine with Prism's and Epmc's explicit engines, respectively. We observe that in both cases, Storm outperforms the competing tools for most instances. In the comparison with Prism, the highest speedups are achieved for CTMCs and PA and for DTMCs the tools tend to be on par. The points on the "error" line are mostly the result of Prism's



(a) Storm's sparse engine vs. Prism's explicit engine.



(b) Storm's sparse engine vs. Epmc's explicit engine.

Figure 7.3: Comparison of explicit-state model checking.

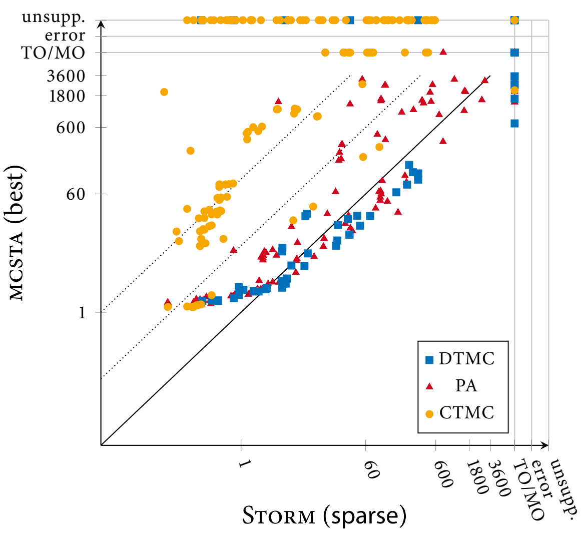


Figure 7.4: STORM's sparse engine vs. MCSTA's best engine.

explicit engine not supporting some of the reward objectives. When comparing with EPMC, it is the CTMCs for which the tools behave comparably and for the other model classes STORM tends to be faster. However, we notice that EPMC produces 8 incorrect results (only DTMCs and PA), which is responsible for most points on the "error" line. Figure 7.4 shows the comparison of the sparse engine of STORM with the best (for each individual verification task) of the three engines of MCSTA. Overall, the model checkers perform similarly. MCSTA is typically faster on DTMCs since for these models the model building times often outweigh the verification times and MCSTA's model building step is roughly twice as fast as STORM's (in the sparse engine). Conversely, STORM has advantages when it comes to (i) CTMCs and (ii) models that are harder to solve such as some of the larger PA.

Figure 7.5 shows the quantile plots of the performances of the tools. We observe that STORM is able to solve the most instances for any given time and PRISM, EPMC and MCSTA perform similarly.

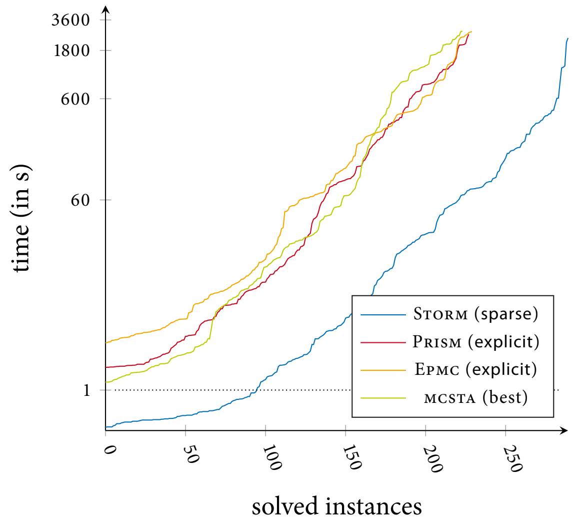


Figure 7.5: Quantile plots comparing the explicit engines of the tools.

7.6.4 Symbolic Probabilistic Model Checking

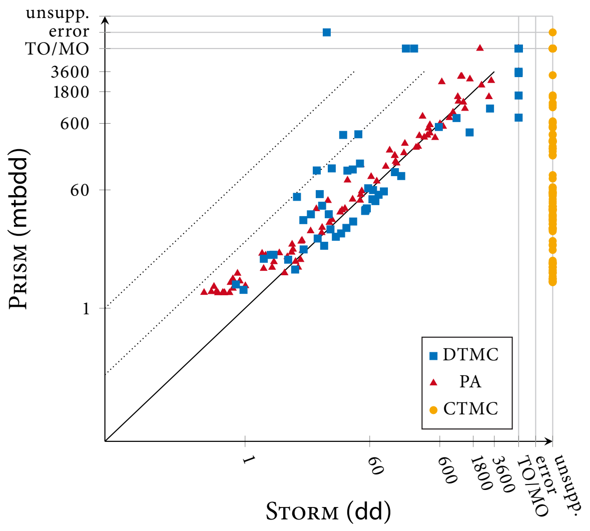
Now, we compare the symbolic engines of the tools. By symbolic we refer to approaches that perform the model building and verification steps mostly on MTBDDs. More specifically, these are

» Prism's hybrid and mtbdd engines and

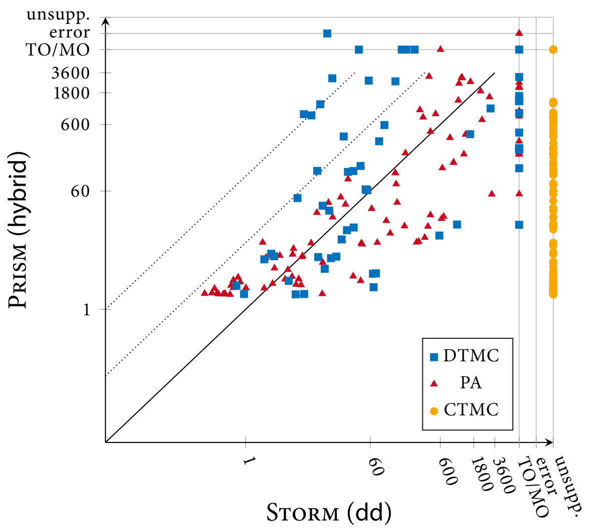
» Storm's dd engine.

Although the hybrid engine of Prism makes some of the information in the MTBDDs explicit, regarding memory requirements it is similar to fully DD-based verification and we therefore consider it here.

Figure 7.6 shows the scatter plots of the experimental results. When comparing Storm's dd engine with Prism's dd engine, we see that they show similar behavior except for a few outliers. This picture becomes different when comparing with Prism's hybrid engine. Here, we see a lot of variance but no clear trend. We can see that for some



(a) STORM's dd engine vs. PRISM's mtbdd engine.



(b) STORM's dd engine vs. PRISM's hybrid engine.

Figure 7.6: Comparison of symbolic model checking.

models, PRISM's hybrid clearly significantly improved runtimes whereas for (a few) other it decreased performance.

In Figure 7.7 we show the number of solved instances in a given time limit, once with and once without CTMCs since they are not supported by STORM's dd engine. In the former case, PRISM solves significantly more instances within the time limit with either engine, but there are clear (runtime) advantages for its hybrid engine. When disregarding the unsupported (by STORM) CTMCs, the tools perform similarly, but ultimately PRISM hybrid outperforms STORM dd.

7.6.5 Hybrid Probabilistic Model Checking

Next, we compare the engines of the tools that make some (or even all) of the information stored in MTBDDs explicit during the numerical solution phase. Even though we have considered it for the symbolic comparison as well, we also count PRISM's hybrid engine to this category. Therefore, we now consider:

» PRISM's sparse and hybrid engines,

» EPMC's dd engine, and

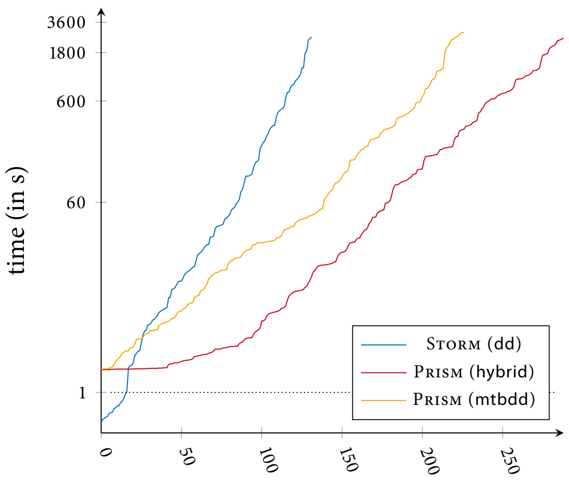
» STORM's hybrid engine.

We give the scatter plots comparing STORM with PRISM's two engines in Figure 7.8. In both comparisons, the majority of instances are solved quicker by STORM than by PRISM. The comparison of STORM and EPMC's dd engine is depicted in Figure 7.9(a). Again, STORM solves most instances faster. We remark that EPMC returned incorrect results on 23 verification tasks, which causes most of the "error" results.

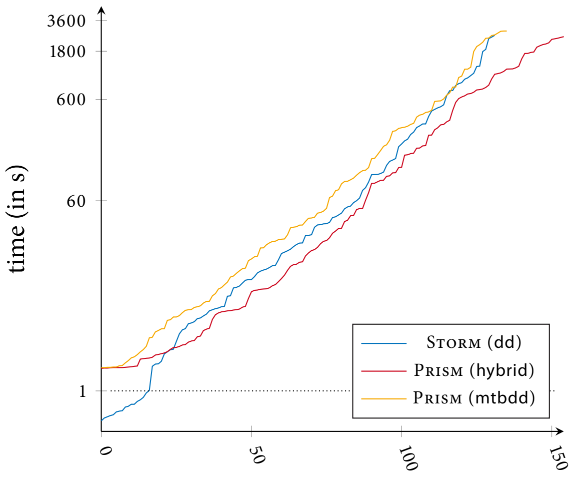
Finally, Figure 7.9(b) plots the performances of all engines of this category in which STORM consistently solves the most instances in any given time limit.

7.6.6 Best Engine

Finally, we compare the tools' best engines for each verification task. Even in the non-probabilistic setting, many tools have a variety of engines and are typically referred to as *portfolio solvers*. Assuming a moderate number of such engines, their strengths can in practice be combined without knowing which engine is best suited for a particular instance by running them in parallel. In this spirit, [Dem+17] suggests that heuristics may be used to build well-performing portfolio solvers. We therefore argue that also in

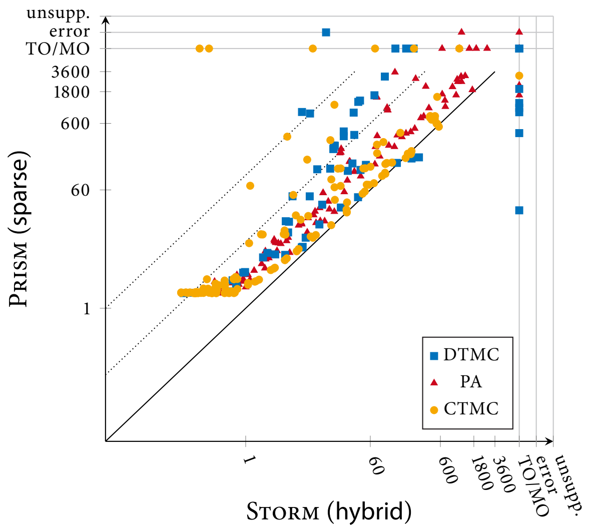


(a) With CTMCs.

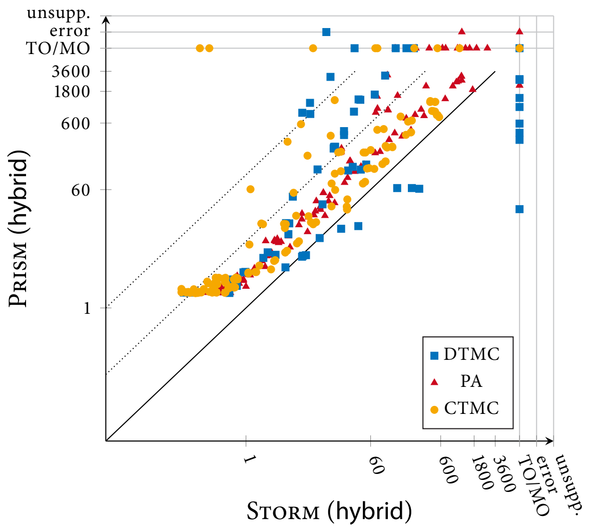


(b) Without CTMCs.

Figure 7.7: Quantile plots comparing the symbolic engines of the tools.

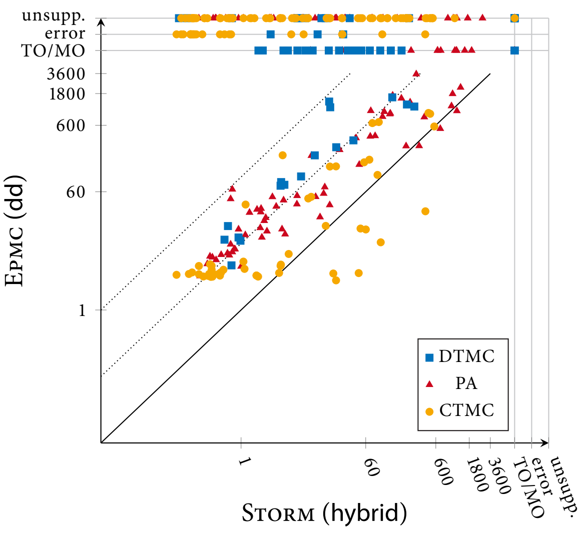


(a) STORM's hybrid engine vs. PRISM's sparse engine.

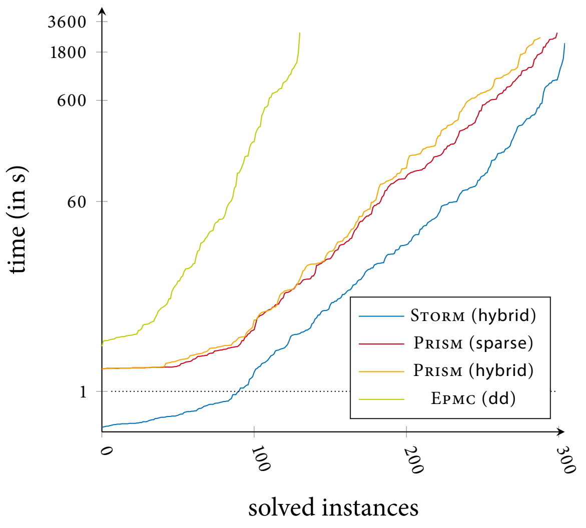


(b) STORM's hybrid engine vs. PRISM's hybrid engine.

Figure 7.8: Comparison of hybrid model checking.



(a) Storm's hybrid engine vs. Epmc's dd engine.



(b) Quantile plot comparing the hybrid engines of the tools.

Figure 7.9: Comparison of hybrid engines.

the probabilistic setting it makes sense to compare the performances of the tools when selecting the best engine for each specific verification task.

Figure 7.10(a) shows that Storm is faster than Prism in the majority of cases. Storm solves some of the verification tasks one to two orders of magnitude faster. The same observations can be made when comparing Storm with Epmc's best engine (see Figure 7.10(b)). Here, we also see that Epmc can solve a number of CTMCs faster than Storm. At the same time, however, Epmc returns incorrect results for other CTMCs. Finally, Figure 7.11 shows the comparison of Storm and mcsta. For the instances that can be solved within less than (around) 500 seconds, the tools overall perform rather similarly. For the larger PA instances, we see that Storm tends to solve them roughly one order of magnitude faster.

Figure 7.12 summarizes the performances of the best engines of the tools: *Storm solves more instances in a given time than any other competitor.*

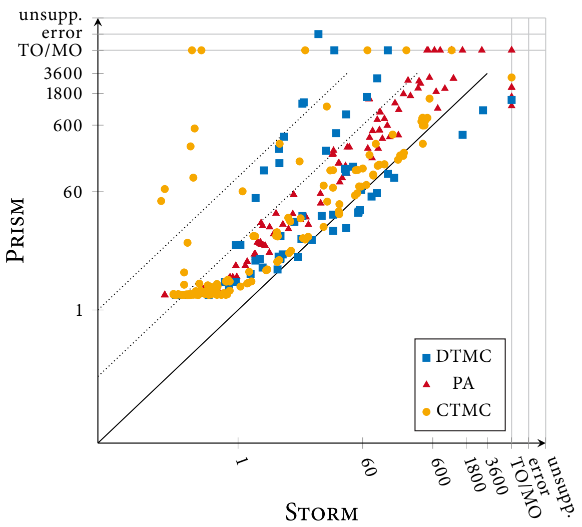
7.6.7 Sound Model Checking

We already mentioned several times that the tools technically do not give guarantees concerning the accuracy of (most of) their results even when assuming the absence of bugs. One of the fundamental reasons for this are the iterative techniques like value iteration that are used to compute probabilities and rewards. Recent work [HM14; Bai+17b; QK18] proposes to address this problem by bounding the target value from below *and above* and therefore guarantee that the actual solution lies within a certain interval. Storm, Prism and mcsta offer "sound" modes based on the aforementioned algorithms that achieve the target precision (relative $\epsilon = 10^{-6}$). More specifically, Prism and mcsta offer *interval iteration* in all of their engines whereas Storm supports *sound value iteration* in the sparse and hybrid engines.

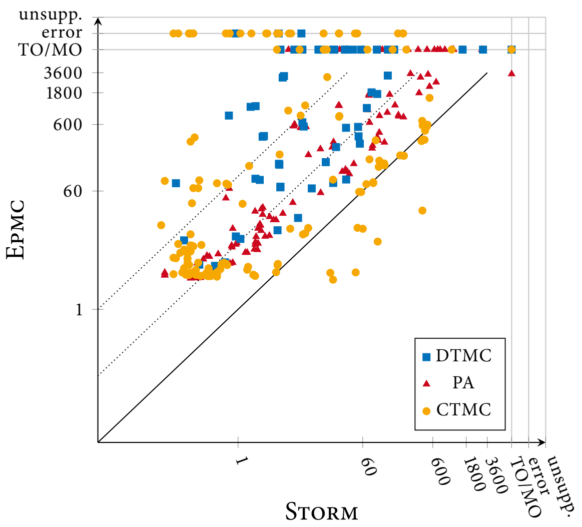
Figure 7.13 shows the experimental results when selecting the sound techniques and the best engine for each instance for all three tools. Except for few instances, Storm outperforms both Prism and mcsta and can solve more instances within the resource limit. This is manifested in the corresponding quantile plot in Figure 7.14.

7.6.8 Exact Model Checking

Another source of inaccuracies is the use of floating point arithmetic [Wim+08; Bau+17]. In certain cases, for example the presence of very small probabilities or nested specifications with precise bounds, it may not be enough to use the techniques referred to in Section 7.6.7. Instead, it may be necessary to compute probabilities and rewards *exactly*. Prism and Storm offer to compute exact values for many types of queries. In



(a) Storm's best engine vs. Prism's best engine.



(b) Storm's best engine vs. Epmc's best engine.

Figure 7.10: Comparison of best engines.

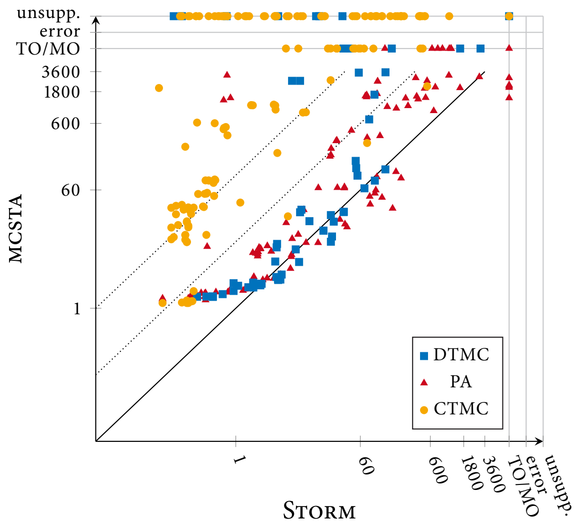


Figure 7.11: Storm's best engine vs. mcsta's best engine.

Figure 7.15(a), we compare the "exact" modes modes of both tools building on their explicit-state engines (sparse for Storm, (a parametric extension of) explicit for Prism). We see that Storm significantly outperforms Prism and achieves speedups of more than two orders of magnitude for many examples, in particular on CTMCs and PA.

In fact, in contrast to Prism, Storm can also store rational numbers in MTBDDs, which entails that Storm's hybrid engine can also compute exact results. In Figure 7.15(b) we compare both of Storm's "exact" engines with Prism's exact mode and see that Storm can solve more than twice as many instances within the resource limit.

7.7 Conclusion

Six years ago, we started to build a new probabilistic model checker from scratch. Today, Storm can determine a broad range of interesting measures on models involving nondeterminism, randomization and (randomized) timing. By offering multiple engines, it can efficiently treat a wide range of input models. Storm goes beyond "simple" probabilistic model checking in numerous ways such as parameter synthesis, permissive scheduler synthesis, counterexample generation, and the computation of reachability

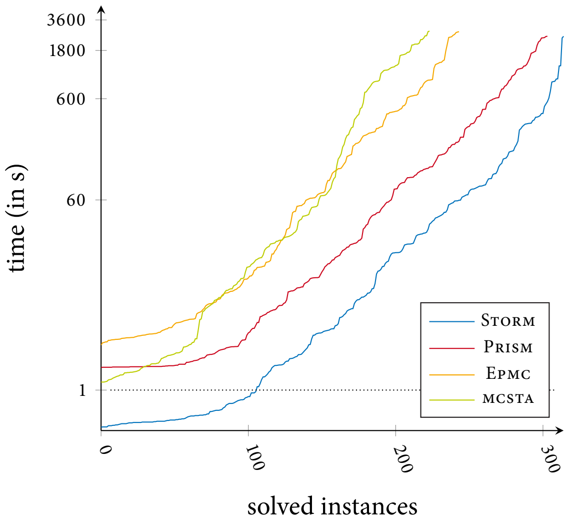
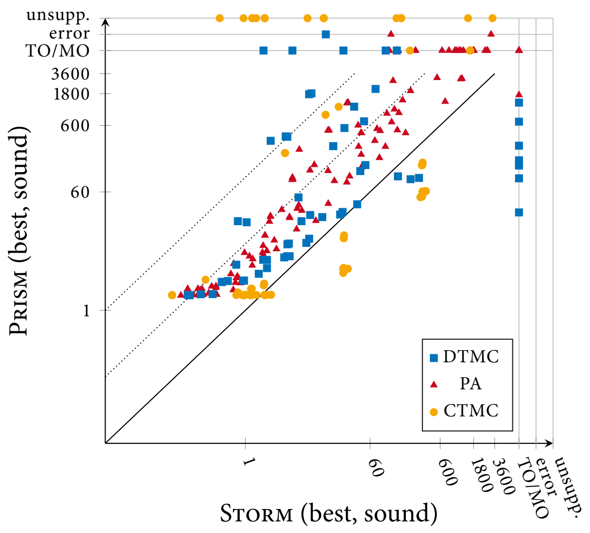


Figure 7.12: Quantile plots comparing the best engines of the tools.

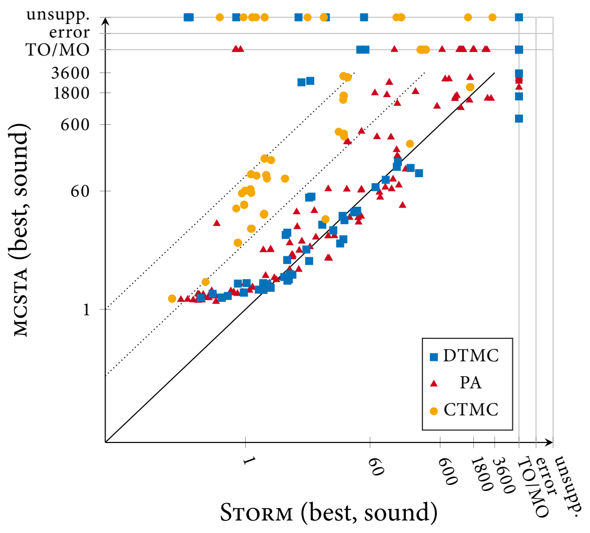
probabilities in infinite-state PA. From a developer point of view, Storm offers abstract solver interfaces with various concrete implementations. This makes it easy to reuse high-performance libraries as well as plugging in new solvers. Through its **Python** API, Storm allows rapid prototyping using high performance routines.

In an extensive evaluation with three other state-of-the-art probabilistic model checkers, we showed that Storm's performance compares favorably for many instances from the well-known Prism benchmark suite across all verification approaches. This lead becomes even clearer when the verification result needs to achieve a specified precision or be exact.

However, as ever so often is the case, there is no "one-size-fits-all" solution in probabilistic model checking and it ultimately depends on the problem at hand which approach and tool will deliver the best results. All of the state-of-the-art model checkers have unique feature sets that come with certain strengths and weaknesses. We believe Storm's characteristics to be unique in the current model checker landscape.



(a) Storm's best engine (sound mode) vs. Prism's best engine (sound mode).



(b) Storm's best engine (sound mode) vs. mcsta's best engine (sound mode).

Figure 7.13: Comparison of the sound modes.

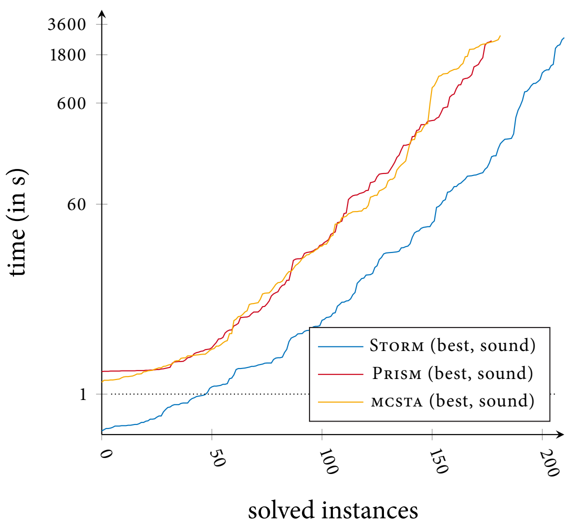
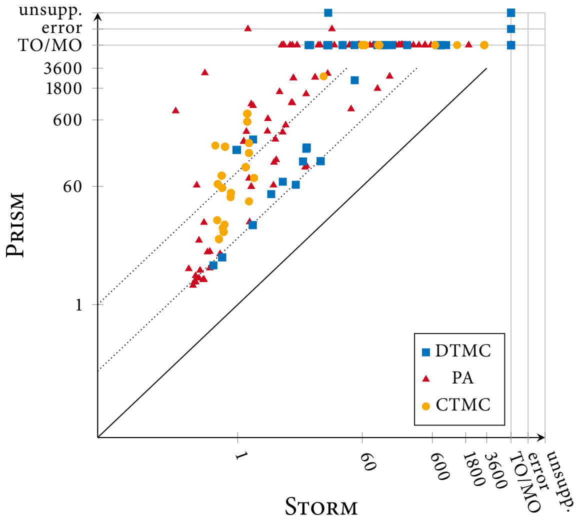
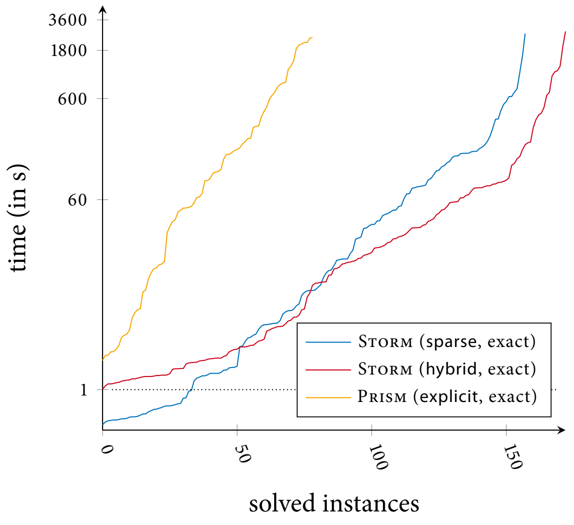


Figure 7.14: Quantile plot comparing the sound modes of the tools.



(a) Storm's exact mode vs. Prism's exact mode.



(b) Quantile plot comparing the exact modes of the tools.

Figure 7.15: Comparison of the exact modes of Storm and Prism.

Chapter 8

Conclusion

The fundamental challenge concerning the scalability of probabilistic model checking is the state space explosion problem. Through the curse of dimensionality, systems quickly become too large to be effectively analyzable. Symbolic methods try to exploit the structure of the model by considering *sets* of states and transitions rather than individual entities. They often obtain succinct representations, because of the structured nature of many models.

In this thesis, we have covered the symbolic treatment of models exhibiting nondeterministic, probabilistic, and randomly timed behavior. In Chapter 3, we showed how the **JANI** language allows to succinctly and in a structured manner cover a wide range of models and yet is easy to parse and extend. To substantiate the attempt **JANI** makes at unifying a divided tool landscape, we formalized the semantics of the fragment corresponding to Markov reward automata.

Counterexamples are key to increasing the acceptance of formal methods among engineers. Chapter 4 presented an approach to synthesize high-level counterexamples in terms of **JANI** (sub)specifications. Through a smart enumeration of solutions to a satisfiability problem and by leveraging the high performance of off-the-shelf probabilistic model checkers, we obtained speedups of several orders of magnitude and were able to treat models with millions of states and decisions

In Chapter 5 we presented how symbolic bisimulation minimization combines the strengths of decision diagrams for the compact representation of probabilistic systems and and bisimulation minimization. We extended previously existing approaches to efficiently deal with cost structures and nondeterminism and speed up the model extraction step. In an extensive evaluation, we showed that bisimulation minimization can reduce the sizes of models drastically.

To push the boundaries regarding the model sizes even further, we discussed how to automatically tailor abstractions to specific models and target objectives in Chapter 6. We showed how stochastic games can serve as a representation of the abstraction and be extracted directly from a high-level specification in **JANI**. By analyzing these abstractions, we obtained both lower and upper bounds on reachability probabilities in infinite probabilistic automata. Apart from amending the refinement procedure of previous work, we discuss crucial optimizations and show that our implementation is competitive and the only one that is able to achieve strictly sound bounds.

Finally, we presented Storm, a modern probabilistic model checker. It features a wide range of input formalisms and verification approaches and goes beyond standard model checking queries by, for instance, supporting the synthesis of counterexamples (as above) and permissive schedulers. In particular, all the techniques developed in this thesis have been integrated in Storm. Together with its strong focus on performance, this makes Storm's feature set unique in the current tool landscape. A performance comparison with three other state-of-the-art probabilistic model checkers revealed that Storm performs favorably across all supported model classes and verification approaches.

Despite the major advances in the field, more than enough challenges remain to be tackled. Since we already included detailed descriptions of further work in the individual chapters, we refrain from restating them here. Rather, we point out two general directions of research that appear to be promising to us.

A few years ago, IC3 [Bra11] constituted a breakthrough for the verification of hardware circuits. By formulating the problem as a series of small satisfiability problems, it benefits directly from the substantial improvements of solvers over the past decade. Substantial research effort has been spent on adapting the approach to software model checking [CG12; LNN15; GLW16]. The scalability of probabilistic model checking could potentially be pushed significantly through a similar approach, but ultimately it remains unclear how to move all crucial aspects of the algorithm to the probabilistic realm.

Finally, to better assess the strengths and weaknesses of the state-of-the-art, we feel that more research regarding the applicability of probabilistic model checking in industrial scenarios should be conducted. While improving the performance of algorithms and tools using benchmark models is certainly valuable, real-world models beyond the academic context could help identifying more important shortcomings and bottlenecks that in turn may steer future work more efficiently. We hope that **JANI** improves this process by providing a common modeling and interfacing language among tools.

Bibliography

[96] *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. Standard. 1996 (cit. on p. 46).

[AB17] John Abbott and Anna Maria Bigatti. "CoCoA-5.2.2 and CoCoALib". In: *ACM Comm. Computer Algebra* 51.3 (2017), pp. 95–97 (cit. on p. 237).

[Ábr+14] Erika Ábrahám, Bernd Becker, Christian Dehnert, Nils Jansen, Joost-Pieter Katoen, and Ralf Wimmer. "Counterexample Generation for Discrete-Time Markov Models: An Introductory Survey". In: *Formal Methods for Executable Software Models - 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2014, Bertinoro, Italy, June 16-20, 2014, Advanced Lectures*. Vol. 8483. 2014, pp. 65–121.

[ABR13] D. E. Nadales Agut, D. A. van Beek, and J. E. Rooda. "Syntax and semantics of the compositional interchange format for hybrid systems". In: *J. Log. Algebr. Program.* 82.1 (2013), pp. 1–52 (cit. on p. 80).

[AH99] Rajeev Alur and Thomas A. Henzinger. "Reactive Modules". In: *Formal Methods in System Design* 15.1 (1999), pp. 7–48 (cit. on p. 74).

[AL10] Husain Aljazzar and Stefan Leue. "Directed Explicit State-Space Search in the Generation of Counterexamples for Stochastic Model Checking". In: *IEEE Trans. Software Eng.* 36.1 (2010), pp. 37–60 (cit. on p. 84).

[Amp14] Elvio Gilberto Amparore. "A New GreatSPN GUI for GSPN Editing and CSLTA Model Checking". In: *Quantitative Evaluation of Systems - 11th*

International Conference, QEST 2014, Florence, Italy, September 8-10, 2014. Proceedings. Vol. 8657. 2014, pp. 170–173 (cit. on p. 78).

[AR07] Luca de Alfaro and Pritam Roy. “Magnifying-Lens Abstraction for Markov Decision Processes”. In: *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*. Vol. 4590. 2007, pp. 325–338 (cit. on p. 166).

[Bab+15] Tomás Babiak, Frantisek Blahoudek, Alexandre Duret-Lutz, Joachim Klein, Jan Kretınský, David Müller, David Parker, and Jan Strejcek. “The Hanoi Omega-Automata Format”. In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Vol. 9206. 2015, pp. 479–486 (cit. on p. 80).

[Bai+05] Christel Baier, Joost-Pieter Katoen, Holger Hermanns, and Verena Wolf. “Comparative branching-time semantics for Markov chains”. In: *Inf. Comput.* 200.2 (2005), pp. 149–214 (cit. on p. 165).

[Bai+14] Christel Baier, Joachim Klein, Sascha Klüppelholz, and Steffen Märcker. “Computing Conditional Probabilities in Markovian Models Efficiently”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*. Vol. 8413. 2014, pp. 515–530 (cit. on pp. 235, 241).

[Bai+17a] Christel Baier, Joachim Klein, Sascha Klüppelholz, and Sascha Wunderlich. “Maximizing the Conditional Expected Reward for Reaching the Goal”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*. Vol. 10206. 2017, pp. 269–285 (cit. on pp. 235, 241).

[Bai+17b] Christel Baier, Joachim Klein, Linda Leuschner, David Parker, and Sascha Wunderlich. “Ensuring the Reliability of Your Model Checker: Interval Iteration for Markov Decision Processes”. In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*. Vol. 10426. 2017, pp. 160–180 (cit. on pp. 199, 215, 226, 250, 260).

[Bai98] Christel Baier. “On algorithmic verification methods for probabilistic systems”. habilitation. Fakultät für Mathematik und Informatik,Universität Mannheim, 1998 (cit. on p. 36).

[Bau+17] Matthew S. Bauer, Umang Mathur, Rohit Chadha, A. Prasad Sistla, and Mahesh Viswanathan. “Exact quantitative probabilistic model checking through rational search”. In: *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*. 2017, pp. 92–99 (cit. on pp. 243, 260).

[BB87] Tommaso Bolognesi and Ed Brinksma. “Introduction to the ISO Specification Language LOTOS”. In: *Computer Networks* 14 (1987), pp. 25–59 (cit. on p. 80).

[BDM16] Carlos E. Budde, Pedro D’Argenio, and Raúl E. Monti. “Compositional Construction of Importance Functions in Fully Automated Importance Splitting”. In: *10th EAI International Conference on Performance Evaluation Methodologies and Tools, VALUETOOLS 2016, Taormina, Italy, 25th-28th Oct 2016*. 2016 (cit. on pp. 75, 80).

[Bee+14] D. A. van Beek, Wan Fokkink, D. Hendriks, A. Hofkamp, Jasen Markovski, J. M. van de Mortel-Fronczak, and Michel A. Reniers. “CIF 3: Model-Based Engineering of Supervisory Controllers”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*. Vol. 8413. 2014, pp. 575–580 (cit. on p. 80).

[Beh+06] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Håkansson, Paul Pettersson, Wang Yi, and Martijn Hendriks. “UPPAAL 4.0”. In: *Third International Conference on the Quantitative Evaluation of Systems (QEST 2006), 11-14 September 2006, Riverside, California, USA*. 2006, pp. 125–126 (cit. on pp. 48, 85).

[Bel57] Richard Bellman. “A Markovian decision process”. In: *Journal of Mathematics and Mechanics* (1957), pp. 679–684 (cit. on p. 21).

[Bel58] Richard Bellman. “Dynamic Programming and Stochastic Control Processes”. In: *Information and Control* 1.3 (1958), pp. 228–239 (cit. on p. 41).

[BEM00] Christel Baier, Bettina Engelen, and Mila E. Majster-Cederbaum. “Deciding Bisimilarity and Similarity for Probabilistic Processes”. In: *J. Comput. Syst. Sci.* 60.1 (2000), pp. 187–231 (cit. on p. 141).

[Ber+18] Philipp Berger, Joost-Pieter Katoen, Erika Ábrahám, Md Tawhid Bin Waez, and Thomas Rambow. “Verifying Auto-generated C Code from Simulink - An Experience Report in the Automotive Domain”. In: *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated*

Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings. Vol. 10951. 2018, pp. 312–328 (cit. on p. 2).

[BFK02] Christian Bauer, Alexander Frink, and Richard Kreckel. "Introduction to the GiNaC Framework for Symbolic Computation within the C++ Programming Language". In: *J. Symb. Comput.* 33.1 (2002), pp. 1–12 (cit. on p. 237).

[BFT15] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.5*. Tech. rep. www.smt-lib.org. Dep. of Computer Science,The University of Iowa, 2015 (cit. on pp. 81, 248).

[Bie+03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. "Bounded model checking". In: *Advances in Computers* 58 (2003), pp. 117–148 (cit. on p. 111).

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008 (cit. on pp. 3, 24, 25).

[BK11] Dirk Beyer and M. Erkan Keremoglu. "CPAchecker: A Tool for Configurable Software Verification". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Vol. 6806. 2011, pp. 184–190 (cit. on p. 83).

[BLR11] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. "A decade of software model checking with SLAM". In: *Commun. ACM* 54.7 (2011), pp. 68–76 (cit. on pp. 3, 83).

[BO05] Stefan Blom and Simona Orzan. "Distributed state space minimization". In: *STTT* 7.3 (2005), pp. 280–291 (cit. on p. 112).

[Boh+06] Henrik C. Bohnenkamp, Pedro R. D'Argenio, Holger Hermanns, and Joost-Pieter Katoen. "MODEST: A Compositional Modeling Formalism for Hard and Softly Timed Systems". In: *IEEE Trans. Software Eng.* 32.10 (2006), pp. 812–830 (cit. on pp. 47, 48, 76).

[Boh14] Dimitri Bohlender. "Accelerating Predicate Abstraction for Probabilistic Automata". MA thesis. RWTH Aachen University, 2014 (cit. on pp. 6, 9, 166).

[Brá+14] Tomás Brázdil, Krishnendu Chatterjee, Martin Chmelik, Vojtech Forejt, Jan Kretınský, Marta Z. Kwiatkowska, David Parker, and Mateusz Ujma. "Verification of Markov Decision Processes Using Learning Algorithms". In: *Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings*. Vol. 8837. 2014, pp. 98–114 (cit. on p. 241).

[Brá+15] Tomás Brázdil, Krishnendu Chatterjee, Martin Chmelik, Andreas Fellner, and Jan Kretınský. “Counterexample Explanation by Learning Small Strategies in Markov Decision Processes”. In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Vol. 9206. 2015, pp. 158–177 (cit. on p. 85).

[Bra11] Aaron R. Bradley. “SAT-Based Model Checking without Unrolling”. In: *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*. Vol. 6538. 2011, pp. 70–87 (cit. on pp. 111, 268).

[Bra14] Tim Bray. “The JavaScript Object Notation (JSON) Data Interchange Format”. In: *RFC* 7159 (2014), pp. 1–16 (cit. on p. 46).

[BS92] Amar Bouali and Robert de Simone. “Symbolic Bisimulation Minimisation”. In: *Computer Aided Verification, Fourth International Workshop, CAV ’92, Montreal, Canada, June 29 - July 1, 1992, Proceedings*. Vol. 663. 1992, pp. 96–108 (cit. on p. 147).

[Bud+17] Carlos E. Budde, Christian Dehnert, Ernst Moritz Hahn, Arnd Hartmanns, Sebastian Junges, and Andrea Turrini. “JANI: Quantitative Model and Tool Interaction”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*. Vol. 10206. 2017, pp. 151–168 (cit. on pp. 5, 9, 46, 48, 73).

[Can+08] Ran Canetti, Ling Cheung, Dilsun Kirli Kaynar, Moses Liskov, Nancy A. Lynch, Olivier Pereira, and Roberto Segala. “Analyzing Security Protocols Using Time-Bounded Task-PIOAs”. In: *Discrete Event Dynamic Systems* 18.1 (2008), pp. 111–159 (cit. on p. 85).

[CC77] Patrick Cousot and Radhia Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. 1977, pp. 238–252 (cit. on pp. 12, 170).

[CC92] Patrick Cousot and Radhia Cousot. “Abstract Interpretation and Application to Logic Programs”. In: *J. Log. Program.* 13.2&3 (1992), pp. 103–179 (cit. on p. 172).

[CCD14] Krishnendu Chatterjee, Martin Chmelik, and Przemyslaw Daca. "CEGAR for Qualitative Analysis of Probabilistic Systems". In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. Vol. 8559. 2014, pp. 473–490 (cit. on p. 85).

[CE81] Edmund M. Clarke and E. Allen Emerson. "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic". In: *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*. Vol. 131. 1981, pp. 52–71 (cit. on p. 25).

[CG12] Alessandro Cimatti and Alberto Griggio. "Software Model Checking via IC3". In: *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. Vol. 7358. 2012, pp. 277–293 (cit. on p. 268).

[Cim+02] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. "NuSMV 2: An OpenSource Tool for Symbolic Model Checking". In: *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*. Vol. 2404. 2002, pp. 359–364 (cit. on p. 83).

[Cim+13] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. "The MathSAT5 SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. Vol. 7795. 2013, pp. 93–107 (cit. on pp. 214, 237).

[Cla+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-Guided Abstraction Refinement". In: *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*. Vol. 1855. 2000, pp. 154–169 (cit. on p. 83).

[Cla08] Edmund M. Clarke. "The Birth of Model Checking". In: *25 Years of Model Checking - History, Achievements, Perspectives*. Vol. 5000. 2008, pp. 1–26 (cit. on p. 4).

[Con90] Anne Condon. "On Algorithms for Simple Stochastic Games". In: *Advances In Computational Complexity Theory, Proceedings of a DIMACS Workshop, New Jersey, USA, December 3-7, 1990*. Vol. 13. 1990, pp. 51–72 (cit. on pp. 166, 199, 205).

[Con92] Anne Condon. “The Complexity of Stochastic Games”. In: *Inf. Comput.* 96.2 (1992), pp. 203–224 (cit. on p. 174).

[Cor+15] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. “SMT-RAT: An Open Source C++ Toolbox for Strategic and Parallel SMT Solving”. In: *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*. Vol. 9340. 2015, pp. 360–368 (cit. on p. 237).

[Cou+09] Tod Courtney, Shravan Gaonkar, Ken Keefe, Eric Rozier, and William H. Sanders. “Möbius 2.3: An extensible tool for dependability, security, and performance evaluation of large and complex system models”. In: *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009*. 2009, pp. 353–358 (cit. on p. 82).

[CV03] Edmund M. Clarke and Helmut Veith. “Counterexamples Revisited: Principles, Algorithms, Applications”. In: *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*. Vol. 2772. 2003, pp. 208–224 (cit. on p. 83).

[CV10] Rohit Chadha and Mahesh Viswanathan. “A counterexample-guided abstraction-refinement framework for markov decision processes”. In: *ACM Trans. Comput. Log.* 12.1 (2010), 1:1–1:49 (cit. on p. 166).

[DAr+01] Pedro R. D’Argenio, Bertrand Jeannet, Henrik Ejersbo Jensen, and Kim Guldstrand Larsen. “Reachability Analysis of Probabilistic Systems by Successive Refinements”. In: *Process Algebra and Probabilistic Methods, Performance Modeling and Verification: Joint International Workshop, PAPM-PROBMIV 2001, Aachen, Germany, September 12-14, 2001, Proceedings*. Vol. 2165. 2001, pp. 39–56 (cit. on p. 166).

[DAr+02] Pedro R. D’Argenio, Bertrand Jeannet, Henrik Ejersbo Jensen, and Kim Guldstrand Larsen. “Reduction and Refinement Strategies for Probabilistic Analysis”. In: *Process Algebra and Probabilistic Methods, Performance Modeling and Verification, Second Joint International Workshop PAPM-PROBMIV 2002, Copenhagen, Denmark, July 25-26, 2002, Proceedings*. Vol. 2399. 2002, pp. 57–76 (cit. on p. 166).

[Daw04] Conrado Daws. “Symbolic and Parametric Model Checking of Discrete-Time Markov Chains”. In: *Theoretical Aspects of Computing - ICTAC 2004, First International Colloquium, Guiyang, China, September 20-24, 2004, Revised Selected Papers*. Vol. 3407. 2004, pp. 280–294 (cit. on pp. 244, 247).

[Deh+12] Christian Dehnert, Daniel Gebler, Michele Volpato, and David N. Jansen. "On Abstraction of Probabilistic Systems". In: *Stochastic Model Checking. Rigorous Dependability Analysis Using Model Checking Techniques for Stochastic Systems - International Autumn School, ROCKS 2012, Vahrn, Italy, October 22-26, 2012, Advanced Lectures*. Vol. 8453. 2012, pp. 87–116.

[Deh+14] Christian Dehnert, Nils Jansen, Ralf Wimmer, Erika Ábrahám, and Joost-Pieter Katoen. "Fast Debugging of PRISM Models". In: *Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings*. Vol. 8837. 2014, pp. 146–162 (cit. on pp. 5, 9).

[Deh+15] Christian Dehnert, Sebastian Junges, Nils Jansen, Florian Corzilius, Matthias Volk, Harold Bruintjes, Joost-Pieter Katoen, and Erika Ábrahám. "PROPhESY: A PRObabilistic ParamEter SYnthesis Tool". In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Vol. 9206. 2015, pp. 214–231 (cit. on pp. 163, 244, 248).

[Deh+16] Christian Dehnert, Sebastian Junges, Nils Jansen, Florian Corzilius, Matthias Volk, Joost-Pieter Katoen, Erika Ábrahám, and Harold Bruintjes. "Parameter Synthesis for Probabilistic Systems". In: *19th GI/ITG/GMM Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, MBMV 2016, Freiburg im Breisgau, Germany, March 1-2, 2016*. 2016, pp. 72–74.

[Deh+17] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. "A Storm is Coming: A Modern Probabilistic Model Checker". In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*. Vol. 10427. 2017, pp. 592–600 (cit. on pp. 6, 9, 80).

[Deh11] Christian Dehnert. "Symbolic Bisimulation Minimization for Markov Models". MA thesis. RWTH Aachen University, 2011 (cit. on p. 138).

[Dem+17] Yulia Demyanova, Thomas Pani, Helmut Veith, and Florian Zuleger. "Empirical software metrics for benchmarking of verification tools". In: *Formal Methods in System Design* 50.2-3 (2017), pp. 289–316 (cit. on pp. 240, 256).

[Det+14] Morgan Deters, Andrew Reynolds, Tim King, Clark W. Barrett, and Cesare Tinelli. "A tour of CVC4: How it works, and how to use it". In: *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*. 2014, p. 7 (cit. on p. 237).

[Dij16] Tom van Dijk. "Sylvan: multi-core decision diagrams". PhD thesis. University of Twente, Enschede, Netherlands, 2016 (cit. on pp. 30, 150, 160, 215, 237, 242).

[DK05] Pedro R. D'Argenio and Joost-Pieter Katoen. "A theory of stochastic systems part I: Stochastic automata". In: *Inf. Comput.* 203.1 (2005), pp. 1–38 (cit. on p. 75).

[DKP13] Christian Dehnert, Joost-Pieter Katoen, and David Parker. "SMT-Based Bisimulation Minimisation of Markov Models". In: *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*. Vol. 7737. 2013, pp. 28–47 (cit. on p. 165).

[DLL62] Martin Davis, George Logemann, and Donald W. Loveland. "A machine program for theorem-proving". In: *Commun. ACM* 5.7 (1962), pp. 394–397 (cit. on p. 42).

[DLM16] Pedro R. D'Argenio, Matias David Lee, and Raúl E. Monti. "Input/Output Stochastic Automata - Compositionality and Determinism". In: *Formal Modeling and Analysis of Timed Systems - 14th International Conference, FORMATS 2016, Quebec, QC, Canada, August 24-26, 2016, Proceedings*. Vol. 9884. 2016, pp. 53–68 (cit. on p. 75).

[DM06] Alastair F. Donaldson and Alice Miller. "Symmetry Reduction for Probabilistic Model Checking Using Generic Representatives". In: *Automated Technology for Verification and Analysis, 4th International Symposium, ATVA 2006, Beijing, China, October 23-26, 2006*. Vol. 4218. 2006, pp. 9–23 (cit. on p. 119).

[DP18] Tom van Dijk and Jaco van de Pol. "Multi-core symbolic bisimulation minimisation". In: *STTT* 20.2 (2018), pp. 157–177 (cit. on pp. 128, 131, 132, 150).

[DP60] Martin Davis and Hilary Putnam. "A Computing Procedure for Quantification Theory". In: *J. ACM* 7.3 (1960), pp. 201–215 (cit. on p. 42).

[Drä+15] Klaus Dräger, Vojtech Forejt, Marta Z. Kwiatkowska, David Parker, and Mateusz Ujma. "Permissive Controller Synthesis for Probabilistic Systems". In: *Logical Methods in Computer Science* 11.2 (2015) (cit. on p. 245).

[Dut14] Bruno Dutertre. "Yices 2.2". In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. Vol. 8559. 2014, pp. 737–744 (cit. on p. 237).

[Ecl] Eclipse Foundation. *Eclipse Modeling Framework (EMF) website*. (Visited on 01/02/2018) (cit. on p. 82).

[Eis+13] Christian Eisentraut, Holger Hermanns, Joost-Pieter Katoen, and Lijun Zhang. "A Semantics for Every GSPN". In: *Application and Theory of Petri Nets and Concurrency - 34th International Conference, PETRI NETS 2013, Milan, Italy, June 24-28, 2013. Proceedings*. Vol. 7927. 2013, pp. 90–109 (cit. on p. 78).

[Fen+18] Lu Feng, Mahsa Ghasemi, Kai-Wei Chang, and Ufuk Topcu. "Counterexamples for Robotic Planning Explained in Structured Language". In: *CoRR* abs/1803.08966 (2018) (cit. on p. 85).

[FKP10] Lu Feng, Marta Z. Kwiatkowska, and David Parker. "Compositional Verification of Probabilistic Systems Using Learning". In: *QEST 2010, Seventh International Conference on the Quantitative Evaluation of Systems, Williamsburg, Virginia, USA, 15-18 September 2010*. 2010, pp. 133–142 (cit. on p. 167).

[FM06] Zhaohui Fu and Sharad Malik. "On Solving the Partial MAX-SAT Problem". In: *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*. Vol. 4121. 2006, pp. 252–265 (cit. on pp. 42, 100).

[Frä+11] Martin Fränzle, Ernst Moritz Hahn, Holger Hermanns, Nicolás Wolovick, and Lijun Zhang. "Measurability and safety verification for stochastic hybrid systems". In: *Proceedings of the 14th ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2011, Chicago, IL, USA, April 12-14, 2011*. 2011, pp. 43–52 (cit. on p. 47).

[Fri11] Peter Fritzson. "Modelica - A cyber-physical modeling language and the OpenModelica environment". In: *Proceedings of the 7th International Wireless Communications and Mobile Computing Conference, IWCMC 2011, Istanbul, Turkey, 4-8 July, 2011*. 2011, pp. 1648–1653 (cit. on p. 80).

[G+10] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. http://eigen.tuxfamily.org. 2010 (cit. on p. 237).

[Gan+04] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. "DPLL(T): Fast Decision Procedures". In: *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*. Vol. 3114. 2004, pp. 175–188 (cit. on p. 42).

[Gar+13] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. "CADP 2011: a toolbox for the construction and analysis of distributed processes". In: *STTT* 15.2 (2013), pp. 89–107 (cit. on p. 80).

[GHS18] Paul Gainer, Ernst Moritz Hahn, and Sven Schewe. "Accelerated Model Checking of Parametric Markov Chains". In: *CoRR* abs/1805.05672 (2018) (cit. on p. 244).

[GLW16] Henning Günther, Alfons Laarman, and Georg Weissenbacher. "Vienna Verification Tool: IC3 for Parallel Software - (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Vol. 9636. 2016, pp. 954–957 (cit. on p. 268).

[GM07] Paul Gastin and Pierre Moro. "Minimal Counterexample Generation for SPIN". In: *Model Checking Software, 14th International SPIN Workshop, Berlin, Germany, July 1-3, 2007, Proceedings*. Vol. 4595. 2007, pp. 24–38 (cit. on p. 84).

[Göd31] Kurt Gödel. "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I". In: *Monatshefte für Mathematik und Physik* 38.1 (1931), pp. 173–198 (cit. on p. 42).

[Gor+14] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. "Probabilistic programming". In: *Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*. 2014, pp. 167–181 (cit. on p. 77).

[Gou+15] Stijn de Gouw, Jurriaan Rot, Frank S. de Boer, Richard Bubel, and Reiner Hähnle. "OpenJDK's Java.utils.Collection.sort() Is Broken: The Good, the Bad and the Worst Case". In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Vol. 9206. 2015, pp. 273–289 (cit. on p. 2).

[GP12] Gabriella Gigante and Domenico Pascarella. "Formal Methods in Avionic Software Certification: The DO-178C Perspective". In: *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part II*. Vol. 7610. 2012, pp. 205–215 (cit. on p. 2).

[GS97] Susanne Graf and Hassen Saıdi. “Construction of Abstract State Graphs with PVS”. In: *Computer Aided Verification, 9th International Conference, CAV ’97, Haifa, Israel, June 22-25, 1997, Proceedings*. Vol. 1254. 1997, pp. 72–83 (cit. on p. 185).

[Guc+12] Dennis Guck, Tingting Han, Joost-Pieter Katoen, and Martin R. Neuhäußer. “Quantitative Timed Analysis of Interactive Markov Chains”. In: *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*. Vol. 7226. 2012, pp. 8–23 (cit. on p. 238).

[Guc+13] Dennis Guck, Hassan Hatefi, Holger Hermanns, Joost-Pieter Katoen, and Mark Timmer. “Modelling, Reduction and Analysis of Markov Automata”. In: *Quantitative Evaluation of Systems - 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*. Vol. 8054. 2013, pp. 55–71 (cit. on pp. 151, 301, 303).

[Gur16] Inc. Gurobi Optimization. *Gurobi Optimizer Reference Manual*. 2016 (cit. on pp. 100, 237).

[GVV18] Jan Friso Groote, Jao Rivera Verduzco, and Erik P. de Vink. “An Efficient Algorithm to Determine Probabilistic Bisimulation”. In: *Algorithms* 11.9 (2018), p. 131 (cit. on p. 163).

[Hah+10a] Ernst Moritz Hahn, Holger Hermanns, Björn Wachter, and Lijun Zhang. “PARAM: A Model Checker for Parametric Markov Models”. In: *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*. Vol. 6174. 2010, pp. 660–664 (cit. on p. 244).

[Hah+10b] Ernst Moritz Hahn, Holger Hermanns, Björn Wachter, and Lijun Zhang. “PASS: Abstraction Refinement for Infinite Probabilistic Models”. In: *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Vol. 6015. 2010, pp. 353–357 (cit. on pp. 166, 231).

[Hah+13] Ernst Moritz Hahn, Arnd Hartmanns, Holger Hermanns, and Joost-Pieter Katoen. “A compositional modelling and analysis framework for stochastic hybrid systems”. In: *Formal Methods in System Design* 43.2 (2013), pp. 191–232 (cit. on p. 76).

[Hah+14] Ernst Moritz Hahn, Yi Li, Sven Schewe, Andrea Turrini, and Lijun Zhang. “iscasMc: A Web-Based Probabilistic Model Checker”. In: *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*. Vol. 8442. 2014, pp. 312–317 (cit. on pp. 80, 235).

[Hah+15] Ernst Moritz Hahn, Guangyuan Li, Sven Schewe, Andrea Turrini, and Lijun Zhang. “Lazy Probabilistic Model Checking without Determinisation”. In: *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*. Vol. 42. 2015, pp. 354–367 (cit. on p. 238).

[Har+18] Arnd Hartmanns, Sebastian Junges, Joost-Pieter Katoen, and Tim Quatmann. “Multi-cost Bounded Reachability in MDP”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*. Vol. 10806. 2018, pp. 320–339 (cit. on p. 241).

[HG08] Henri Hansen and Jaco Geldenhuys. “Cheap and Small Counterexamples”. In: *Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, Cape Town, South Africa, 10-14 November 2008*. 2008, pp. 53–62 (cit. on p. 84).

[HH14] Arnd Hartmanns and Holger Hermanns. “The Modest Toolset: An Integrated Environment for Quantitative Modelling and Verification”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*. Vol. 8413. 2014, pp. 593–598 (cit. on pp. 76, 80, 82, 236).

[HHB16] Arnd Hartmanns, Holger Hermanns, and Michael Bungert. “Flexible support for time and costs in scenario-aware dataflow”. In: *2016 International Conference on Embedded Software, EMSOFT 2016, Pittsburgh, Pennsylvania, USA, October 1-7, 2016*. 2016, 3:1–3:10 (cit. on p. 79).

[HHP13] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. “Software Model Checking for People Who Love Automata”. In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. Vol. 8044. 2013, pp. 36–52 (cit. on p. 83).

[HJ94] Hans Hansson and Bengt Jonsson. “A Logic for Reasoning about Time and Reliability”. In: *Formal Asp. Comput.* 6.5 (1994), pp. 512–535 (cit. on p. 26).

[HKD09] Tingting Han, Joost-Pieter Katoen, and Berteun Damman. “Counterexample Generation in Probabilistic Model Checking”. In: *IEEE Trans. Software Eng.* 35.2 (2009), pp. 241–257 (cit. on p. 84).

[HM14] Serge Haddad and Benjamin Monmege. “Reachability in MDPs: Refining Convergence of Value Iteration”. In: *Reachability Problems - 8th International Workshop, RP 2014, Oxford, UK, September 22-24, 2014. Proceedings*. Vol. 8762. 2014, pp. 125–137 (cit. on pp. 199, 215, 226, 243, 260).

[HMW09] Thomas A. Henzinger, Maria Mateescu, and Verena Wolf. “Sliding Window Abstraction for Infinite Markov Chains”. In: *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*. Vol. 5643. 2009, pp. 337–352 (cit. on p. 166).

[Hol14] Gerard J. Holzmann. “Mars code”. In: *Commun. ACM* 57.2 (2014), pp. 64–73 (cit. on pp. 3, 83).

[Hol97] Gerard J. Holzmann. “The Model Checker SPIN”. In: *IEEE Trans. Software Eng.* 23.5 (1997), pp. 279–295 (cit. on p. 83).

[How64] Ronald Arthur Howard. *Dynamic programming and Markov processes*. Wiley for The Massachusetts Institute of Technology, 1964 (cit. on pp. 41, 243).

[HSM97] Jifeng He, Karen Seidel, and Annabelle McIver. “Probabilistic Models for the Guarded Command Language”. In: *Sci. Comput. Program.* 28.2-3 (1997), pp. 171–192 (cit. on p. 240).

[HWZ08] Holger Hermanns, Björn Wachter, and Lijun Zhang. “Probabilistic CEGAR”. In: *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*. Vol. 5123. 2008, pp. 162–175 (cit. on pp. 85, 166, 207).

[IEC15] IEC. *IEC 62279: Railway applications - Communication,signalling and processing systems - Software for railway control and protection systems*. Norm. 2015 (cit. on p. 2).

[INR] INRIA, IMAG. *CADP EXP Language*. (Visited on 01/08/2018) (cit. on p. 58).

[ISO11a] ISO. *ISO 26262: Road vehicles — Functional safety*. Norm. 2011 (cit. on p. 2).

[ISO11b] ISO. *ISO 15909-2:2011. High-level Petri nets – Part 2: Transfer format.* 2011 (cit. on p. 78).

[Jah91] Christian Jahl. "The Information Technology Security Evaluation Criteria". In: *Proceedings of the 13th International Conference on Software Engineering, Austin, TX, USA, May 13-17, 1991.* 1991, pp. 306–312 (cit. on p. 2).

[Jan+14] Nils Jansen, Ralf Wimmer, Erika Ábrahám, Barna Zajzon, Joost-Pieter Katoen, Bernd Becker, and Johann Schuster. "Symbolic counterexample generation for large discrete-time Markov chains". In: *Sci. Comput. Program.* 91 (2014), pp. 90–114 (cit. on p. 85).

[Jan+16] Nils Jansen, Christian Dehnert, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Lukas Westhofen. "Bounded Model Checking for Probabilistic Programs". In: *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*. Vol. 9938. 2016, pp. 68–85 (cit. on pp. 307, 309).

[Jr78] Sheldon B. Akers Jr. "Binary Decision Diagrams". In: *IEEE Trans. Computers* 27.6 (1978), pp. 509–516 (cit. on p. 30).

[Jun+16] Sebastian Junges, Nils Jansen, Christian Dehnert, Ufuk Topcu, and Joost-Pieter Katoen. "Safety-Constrained Reinforcement Learning for MDPs". In: *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Vol. 9636. 2016, pp. 130–146 (cit. on p. 245).

[Kan+15] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. "LTSmin: High-Performance Language-Independent Model Checking". In: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Vol. 9035. 2015, pp. 692–707 (cit. on p. 82).

[Kat+07] Joost-Pieter Katoen, Tim Kemna, Ivan S. Zapreev, and David N. Jansen. "Bisimulation Minimisation Mostly Speeds Up Probabilistic Model Checking". In: *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*. Vol. 4424. 2007, pp. 87–101 (cit. on p. 112).

[Kat+10] Mark Kattenbelt, Marta Z. Kwiatkowska, Gethin Norman, and David Parker. "A game-based abstraction-refinement framework for Markov decision processes". In: *Formal Methods in System Design* 36.3 (2010), pp. 246–280 (cit. on pp. 6, 175, 178).

[Kat+11] Joost-Pieter Katoen, Ivan S. Zapreev, Ernst Moritz Hahn, Holger Hermanns, and David N. Jansen. "The ins and outs of the probabilistic model checker MRMC". In: *Perform. Eval.* 68.2 (2011), pp. 90–104 (cit. on pp. 81, 84, 233).

[Kat+12] Joost-Pieter Katoen, Jaco van de Pol, Mariëlle Stoelinga, and Mark Timmer. "A linear process-algebraic format with data for probabilistic automata". In: *Theor. Comput. Sci.* 413.1 (2012), pp. 36–57 (cit. on p. 85).

[Kat+15] Joost-Pieter Katoen, Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, and Federico Olmedo. "Understanding Probabilistic Programs". In: *Correct System Design - Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Oldenburg, Germany, September 8-9, 2015. Proceedings*. Vol. 9360. 2015, pp. 15–32 (cit. on p. 77).

[Kat16] Joost-Pieter Katoen. "The Probabilistic Model Checking Landscape". In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*. 2016, pp. 31–45 (cit. on p. 45).

[Kel+18] Edon Kelmendi, Julia Krämer, Jan Kretınský, and Maximilian Weininger. "Value Iteration for Simple Stochastic Games: Stopping Criterion and Learning Algorithm". In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. Vol. 10981. 2018, pp. 623–642 (cit. on pp. 199, 215, 226).

[Kle+18] Joachim Klein, Christel Baier, Philipp Chrszon, Marcus Daum, Clemens Dubslaff, Sascha Klüppelholz, Steffen Märcker, and David Müller. "Advances in probabilistic model checking with PRISM: variable reordering, quantiles and weak deterministic Büchi automata". In: *STTT* 20.2 (2018), pp. 179–194 (cit. on p. 235).

[Kle52] Stephen Cole Kleene. "Introduction to metamathematics". In: *Amsterdam, Groningen* (1952) (cit. on p. 13).

[KM03] Stephen Kwek and Kurt Mehlhorn. "Optimal search for rationals". In: *Inf. Process. Lett.* 86.1 (2003), pp. 23–26 (cit. on p. 243).

[KNP06a] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. "Game-based Abstraction for Markov Decision Processes". In: *Third International Conference on the Quantitative Evaluation of Systems (QEST 2006), 11-14 September 2006, Riverside, California, USA*. 2006, pp. 157–166 (cit. on p. 166).

[KNP06b] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. "Symmetry Reduction for Probabilistic Model Checking". In: *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*. Vol. 4144. 2006, pp. 234–248 (cit. on p. 119).

[KNP11] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. "PRISM 4.0: Verification of Probabilistic Real-Time Systems". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Vol. 6806. 2011, pp. 585–591 (cit. on pp. 48, 80, 84, 85, 233, 235).

[KNP12] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. "The PRISM Benchmark Suite". In: *Ninth International Conference on Quantitative Evaluation of Systems, QEST 2012, London, United Kingdom, September 17-20, 2012*. 2012, pp. 203–204 (cit. on pp. 74, 101, 151, 248, 299, 301, 307, 313).

[Koc+18] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *CoRR* abs/1801.01203 (2018) (cit. on p. 1).

[KPC12] Anvesh Komuravelli, Corina S. Pasareanu, and Edmund M. Clarke. "Assume-Guarantee Abstraction Refinement for Probabilistic Systems". In: *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. Vol. 7358. 2012, pp. 310–326 (cit. on p. 167).

[KS13] Ken Keefe and William H. Sanders. "Möbius Shell: A Command-Line Interface for Möbius". In: *Quantitative Evaluation of Systems - 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*. Vol. 8054. 2013, pp. 282–285 (cit. on p. 82).

[KS90] Paris C. Kanellakis and Scott A. Smolka. "CCS Expressions, Finite State Processes, and Three Problems of Equivalence". In: *Inf. Comput.* 86.1 (1990), pp. 43–68 (cit. on p. 112).

[KW16] Joost-Pieter Katoen and Hao Wu. "Probabilistic Model Checking for Uncertain Scenario-Aware Data Flow". In: *ACM Trans. Design Autom. Electr. Syst.* 22.1 (2016), 15:1–15:27 (cit. on p. 79).

[Kwi+10] Marta Z. Kwiatkowska, Gethin Norman, David Parker, and Hongyang Qu. "Assume-Guarantee Verification for Probabilistic Systems". In: *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Vol. 6015. 2010, pp. 23–37 (cit. on p. 166).

[LA04] Chris Lattner and Vikram S. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. 2004, pp. 75–88 (cit. on p. 80).

[Lev17] Nancy G. Leveson. "The Therac-25: 30 Years Later". In: *IEEE Computer* 50.11 (2017), pp. 8–11 (cit. on p. 1).

[Lip+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. "Meltdown". In: *CoRR* abs/1801.01207 (2018) (cit. on p. 1).

[LL13] Florian Leitner-Fischer and Stefan Leue. "Probabilistic fault tree synthesis using causality computation". In: *IJCCBS* 4.2 (2013), pp. 119–143 (cit. on p. 84).

[LNN15] Tim Lange, Martin R. Neuhäußer, and Thomas Noll. "IC3 Software Model Checking on Control Flow Automata". In: *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015*. 2015, pp. 97–104 (cit. on p. 268).

[Mär+17] Steffen Märcker, Christel Baier, Joachim Klein, and Sascha Klüppelholz. "Computing Conditional Probabilities: Implementation and Evaluation". In: *Software Engineering and Formal Methods - 15th International Conference, SEFM 2017, Trento, Italy, September 4-8, 2017, Proceedings*. Vol. 10469. 2017, pp. 349–366 (cit. on pp. 235, 248).

[MB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Vol. 4963. 2008, pp. 337–340 (cit. on pp. 100, 237).

[MCB84] Marco Ajmone Marsan, Gianni Conte, and Gianfranco Balbo. "A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems". In: *ACM Trans. Comput. Syst.* 2.2 (1984), pp. 93–122 (cit. on pp. 78, 240).

[McM18] Kenneth L. McMillan. "Interpolation and Model Checking". In: *Handbook of Model Checking.* 2018, pp. 421–446 (cit. on p. 43).

[MM05] Annabelle McIver and Carroll Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer, 2005 (cit. on p. 77).

[NK07] Martin R. Neuhäußer and Joost-Pieter Katoen. "Bisimulation and Logical Preservation for Continuous-Time Markov Decision Processes". In: *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings*. Vol. 4703. 2007, pp. 412–427 (cit. on p. 119).

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999 (cit. on p. 92).

[Par03] David Anthony Parker. "Implementation of symbolic model checking for probabilistic systems". PhD thesis. University of Birmingham, UK, 2003 (cit. on pp. 27, 34, 36, 112, 147, 150, 154).

[Pnu77] Amir Pnueli. "The Temporal Logic of Programs". In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. 1977, pp. 46–57 (cit. on p. 25).

[Pra95] Vaughan R. Pratt. "Anatomy of the Pentium Bug". In: *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22-26, 1995, Proceedings*. Vol. 915. 1995, pp. 97–107 (cit. on p. 1).

[QJK17] Tim Quatmann, Sebastian Junges, and Joost-Pieter Katoen. "Markov Automata with Multiple Objectives". In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*. Vol. 10426. 2017, pp. 140–159 (cit. on pp. 151, 163, 241, 301, 303).

[QK18] Tim Quatmann and Joost-Pieter Katoen. "Sound Value Iteration". In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. Vol. 10981. 2018, pp. 643–661 (cit. on pp. 199, 215, 226, 243, 250, 260).

[Qua+15] Tim Quatmann, Nils Jansen, Christian Dehnert, Ralf Wimmer, Erika Ábrahám, Joost-Pieter Katoen, and Bernd Becker. “Counterexamples for Expected Rewards”. In: *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*. Vol. 9109. 2015, pp. 435–452.

[Qua+16] Tim Quatmann, Christian Dehnert, Nils Jansen, Sebastian Junges, and Joost-Pieter Katoen. “Parameter Synthesis for Markov Models: Faster Than Ever”. In: *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*. Vol. 9938. 2016, pp. 50–67 (cit. on p. 244).

[Qua16] Tim Quatmann. “Multi-Objective Model Checking of Markov Automata”. MA thesis. RWTH Aachen University, 2016 (cit. on p. 24).

[Ric+04] Raymond Richards, David Greve, Matthew Wilding, and W Mark Vanfleet. “The common criteria, formal methods and ACL2”. In: *ACL2 Workshop*. 2004 (cit. on p. 2).

[RS15] Enno Ruijters and Mariëlle Stoelinga. “Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools”. In: *Computer Science Review* 15 (2015), pp. 29–62 (cit. on p. 240).

[San10] Scott Sanner. “Relational Dynamic Influence Diagram Language (RDDL): Language Description”. In: *International Probabilistic Planning Competition (IPPC)* (2010). users.cecs.anu.edu.au/ ssanner/IPPC_2011/RDDL.pdf (cit. on p. 80).

[Saz13] Sergey Sazonov. “Property Preservation under Bisimulations on Markov Automata”. MA thesis. RWTH Aachen University, 2013 (cit. on p. 119).

[SB05] Viktor Schuppan and Armin Biere. “Shortest Counterexamples for Symbolic Model Checking of LTL with Past”. In: *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*. Vol. 3440. 2005, pp. 493–509 (cit. on p. 84).

[Seg95] Roberto Segala. “Modeling and verification of randomized distributed real-time systems”. PhD thesis. Massachusetts Institute of Technology, Cambridge, MA, USA, 1995 (cit. on p. 117).

[Sha53] L. S. Shapley. “Stochastic Games”. In: *Proceedings of the National Academy of Sciences* 39.10 (1953), pp. 1095–1100 (cit. on p. 166).

[Si+16] Xujie Si, Xin Zhang, Vasco M. Manquinho, Mikolás Janota, Alexey Ignatiev, and Mayur Naik. "On Incremental Core-Guided MaxSAT Solving". In: *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*. Vol. 9892. 2016, pp. 473–482 (cit. on p. 42).

[SL95] Roberto Segala and Nancy A. Lynch. "Probabilistic Simulations for Probabilistic Processes". In: *Nordic Journal of Computation* 2.2 (1995), pp. 250–273 (cit. on p. 117).

[SLG09] Jan Stöcker, Frédéric Lang, and Hubert Garavel. "Parallel Processes with Real-Time and Data: The ATLANTIF Intermediate Format". In: *Integrated Formal Methods, 7th International Conference, IFM 2009, Düsseldorf, Germany, February 16-19, 2009. Proceedings*. Vol. 5423. 2009, pp. 88–102 (cit. on p. 81).

[Sol+06] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodık, Sanjit A. Seshia, and Vijay A. Saraswat. "Combinatorial sketching for finite programs". In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*. 2006, pp. 404–415 (cit. on p. 83).

[Som] Fabio Somenzi. *CUDD 3.0.0*. (Visited on 03/11/2018) (cit. on pp. 30, 150, 215, 237, 242).

[Spr00] Jeremy Sproston. "Decidable Model Checking of Probabilistic Hybrid Automata". In: *Formal Techniques in Real-Time and Fault-Tolerant Systems, 6th International Symposium, FTRTFT 2000, Pune, India, September 20-22, 2000, Proceedings*. Vol. 1926. 2000, pp. 31–45 (cit. on p. 47).

[SS96] João P. Marques Silva and Karem A. Sakallah. "GRASP - a new search algorithm for satisfiability". In: *ICCAD*. 1996, pp. 220–227 (cit. on p. 42).

[SZG13] Lei Song, Lijun Zhang, and Jens Chr. Godskesen. "Bisimulations Meet PCTL Equivalences for Probabilistic Automata". In: *Logical Methods in Computer Science* 9.2 (2013) (cit. on p. 119).

[The+06] Bart D. Theelen, Marc Geilen, Twan Basten, Jeroen Voeten, Stefan Valentin Gheorghita, and Sander Stuijk. "A scenario-aware data flow model for combined long-run average and worst-case performance analysis". In: *4th ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2006), 27-29 July 2006, Embassy Suites, Napa, California, USA*. 2006, pp. 185–194 (cit. on p. 79).

[Tim13] Mark Timmer. “Efficient modelling, generation and analysis of Markov automata”. PhD thesis. University of Twente, Enschede, Netherlands, 2013 (cit. on pp. 15, 18, 19, 119).

[Tin02] Cesare Tinelli. “A DPLL-Based Calculus for Ground Satisfiability Modulo Theories”. In: *Logics in Artificial Intelligence, European Conference, JELIA 2002, Cosenza, Italy, September, 23-26, Proceedings*. Vol. 2424. 2002, pp. 308–319 (cit. on p. 42).

[VJK16] Matthias Volk, Sebastian Junges, and Joost-Pieter Katoen. “Advancing Dynamic Fault Tree Analysis - Get Succinct State Spaces Fast and Synthesise Failure Rates”. In: *Computer Safety, Reliability, and Security - 35th International Conference, SAFECOMP 2016, Trondheim, Norway, September 21-23, 2016, Proceedings*. Vol. 9922. 2016, pp. 253–265 (cit. on p. 240).

[Wac11] Björn Wachter. “Refined probabilistic abstraction”. PhD thesis. Saarland University, 2011 (cit. on pp. 6, 166, 171, 172, 174, 182, 191, 201, 202, 205–207, 210, 211, 213, 216, 221).

[WBB09] Ralf Wimmer, Bettina Braitling, and Bernd Becker. “Counterexample Generation for Discrete-Time Markov Chains Using Bounded Model Checking”. In: *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*. Vol. 5403. 2009, pp. 366–380 (cit. on p. 84).

[Wim+08] Ralf Wimmer, Alexander Kortus, Marc Herbstritt, and Bernd Becker. “Probabilistic Model Checking and Reliability of Results”. In: *Proceedings of the 11th IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS 2008), Bratislava, Slovakia, April 16-18, 2008*. 2008, pp. 207–212 (cit. on pp. 215, 226, 243, 260).

[Wim+12] Ralf Wimmer, Nils Jansen, Erika Ábrahám, Bernd Becker, and Joost-Pieter Katoen. “Minimal Critical Subsystems for Discrete-Time Markov Models”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. Vol. 7214. 2012, pp. 299–314 (cit. on p. 84).

[Wim+13] Ralf Wimmer, Nils Jansen, Andreas Vorpahl, Erika Ábrahám, Joost-Pieter Katoen, and Bernd Becker. “High-Level Counterexamples for Probabilistic Automata”. In: *Quantitative Evaluation of Systems - 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*. Vol. 8054. 2013, pp. 39–54 (cit. on p. 5).

[Wim+14] Ralf Wimmer, Nils Jansen, Erika Ábrahám, Joost-Pieter Katoen, and Bernd Becker. "Minimal counterexamples for linear-time probabilistic verification". In: *Theor. Comput. Sci.* 549 (2014), pp. 61–100 (cit. on p. 84).

[Wim+15] Ralf Wimmer, Nils Jansen, Andreas Vorpahl, Erika Ábrahám, Joost-Pieter Katoen, and Bernd Becker. "High-level Counterexamples for Probabilistic Automata". In: *Logical Methods in Computer Science* 11.1 (2015) (cit. on pp. 5, 9, 85, 97, 100, 101).

[Wim10] Ralf Wimmer. "Symbolische Methoden für die probabilistische Verifikation: Zustandsraumreduktion und Gegenbeispiele". PhD thesis. University of Freiburg, 2010 (cit. on pp. 5, 112, 126–128, 131, 132, 145, 147, 151, 158, 243).

[Woo+08] Jim Woodcock, Susan Stepney, David Cooper, John A. Clark, and Jeremy Jacob. "The certification of the Mondex electronic purse to ITSEC Level E6". In: *Formal Asp. Comput.* 20.1 (2008), pp. 5–19 (cit. on p. 2).

[Wus02] Michael Wusk. "ARIES: NASA Langley's Airborne Research Facility". In: *AIAA's Aircraft Technology, Integration, and Operations (ATIO) 2002 Technical Forum*. 2002 (cit. on p. 1).

[WZ10] Björn Wachter and Lijun Zhang. "Best Probabilistic Transformers". In: *Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings.* Vol. 5944. 2010, pp. 362–379 (cit. on p. 166).

[WZH07] Björn Wachter, Lijun Zhang, and Holger Hermanns. "Probabilistic Model Checking Modulo Theories". In: *Fourth International Conference on the Quantitative Evaluaiton of Systems (QEST 2007), 17-19 September 2007, Edinburgh, Scotland, UK*. 2007, pp. 129–140 (cit. on p. 166).

[Zap08] Ivan S. Zapreev. "Model checking Markov chains : techniques and tools". PhD thesis. University of Twente, Enschede, Netherlands, 2008 (cit. on p. 234).

Appendix A

JANI

The full **JANI** specification of the NSMA containing the SMA $\mathfrak{A}_S$ and $\mathfrak{A}_R$ from Chapter 3 is listed below.

```
{
    "jani-version": 1,
    "features": [
        "derived-operators"
    ],
    "name": "nsma",
    "actions": [
        { "name": "reject" },
        { "name": "accept" }
    ],
    "type": "ma",
    "constants": [
         {
             "name": "λ",
             "type": { "base": "real", "kind": "bounded", "lower-bound"
                 : 0 }
         }
     ],
    "automata": [
        {
            "name": "𝔄S",
            "locations": [
                { "name": "wait" },
                {
                    "name": "send",
                    "transient-values": [ { "ref": "p", "value": 1 } ]
                }
            ],
            "initial-locations": [
                "wait"
            ],
            "edges": [
                {
                    "location": "wait",
                    "rate": "λ",
                    "destinations": [ { "location": "send" } ]
                },
                {
                    "location": "send",
                    "action": "send",
                    "destinations": [
                        {
                            "probability": { "exp": { "left": 2, "op":
                                "/", "right": 3 } },
                            "assignments": [ { "ref": "t", "value": 1 }
                                 ]
                        },
                        {
                            "probability": {
                                "exp": { "left": 1, "op": "/", "right":
                                     3 }
                            },
                            "assignments": [
                                { "ref": "t", "value": 2 },
                                {
                                    "ref": "p",
                                    "value": { "exp": { "left": "p", "
                                        op": "+", "right": 1 } },
                                    "index": 1
                                }
                            ]
                        }
                    ]
                }
            ]
        },
        {
            "name": "$\mathfrak{A}_R$",
            "locations": [ { "name": "idle" }, { "name": "busy" } ],
            "initial-locations": [ "idle" ],
            "variables": [ { "name": "m", "type": "int" } ],
            "restrict-initial": { "exp": { "left": "m", "op": "=", "
                right": 0 } },
            "edges": [
                {
                    "location": "idle",
                    "action": "reject",
                    "destinations": [
                        {
                            "assignments": [ { "ref": "p", "value": "3"
                                } ],
                            "location": "idle"
                        }
                    ]
                },
                {
                    "location": "idle",
                    "action": "accept",
                    "destinations": [
                        {
                            "probability": { "exp": { "left": 9, "op":
                                "/", "right": 10 } },
                            "assignments": [ { "ref": "m", "value": "t"
                                , "index": 1 } ],
                            "location": "busy"
                        },
                        {
                            "probability": { "exp": { "left": 1, "op":
                                "/", "right": 10 } },
                            "location": "idle"
                        }
                    ]
                },
                {
                    "location": "busy",
                    "guard": { "exp": { "left": "m", "op": ">", "right"
                        : 0 } },
                    "rate": { "exp": { "left": 1, "op": "/", "right": "
                        m" } },
                    "destinations": [
                        {
                            "assignments": [ { "ref": "m", "value": 0 }
                                ],
                            "location": "idle"
                        }
                    ]
                }
            ]
        }
    ],
    "restrict-initial": {
        "exp": true
    },
    "system": {
        "elements": [ { "automaton": "𝔄_S" },{"automaton": "𝔄_R" }],
        "syncs": [
            {
                "synchronise": [ "send", "accept" ]
            },
            {
                "synchronise": [ "send", "reject" ]
            }
        ]
    },
    "variables": [
        {
            "name": "t",
            "transient": true,
            "initial-value": 0,
            "type": "int"
        },
        {
            "name": "p",
            "transient": true,
            "initial-value": 0,
            "type": "real"
        }
    ]
}
```

Appendix B

Notation

To state the properties used in the experiments, we briefly introduce their syntax. There are three different operators: $\mathsf{P}(\cdot)$, $\mathsf{R}(\cdot)$ and $\mathsf{S}(\cdot)$. $\mathsf{P}(\varphi)$ measures the probability of the contained path formula φ. $\mathsf{R}(\varphi)$ reasons about the reward whose type is described by φ. Finally, $\mathsf{S}(\varphi)$ denotes the long-run (or steady state) probability to be in a state satisfying φ. All operators may be subscripted with a bound on the measure or with "=?" to indicate that the measure is to be computed without a subsequent comparison. For models involving nondeterminism, the superscript may specify how the the nondeterminism is to be resolved ($-$ or $+$) unless that is implicitly given by the bound.

The path formulae that may appear within the probability operator are the usual (bounded or unbounded) until or (bounded or unbounded) eventually modalities and we do not detail them further.

As reward formulae φ we use

» $\mathcal{I}^{=t}$ to denote the expected reward at exactly time step (or point) t,

» $\mathcal{C}^{\leq t}$ to denote the expected reward accumulated until time step (or point) t,

» $\lozenge T$ to denote the expected reward accumulated until reaching the set T, and

» S to denote the expected long-run reward.

If there are multiple reward models associated with a benchmark model, we mention the concrete reward model as an additional subscript of the R operator.

Appendix C

Fast Debugging of JANI Models

All models used for the evaluation in Chapter 4 were taken from the Prism benchmark suite [KNP12] available at

`https://github.com/prismmodelchecker/prism-benchmarks/.`

For every model, we match the parameters with (the names of) the constants of the model file and state the property in the syntax described in Appendix B.

coin. The first parameter of an instance refers to the constant N and the second parameter refers to the constant K. We consider the property

» $\mathsf{P}_{<\lambda}(\Diamond$ "finished" $\wedge$ "all_coins_equal_1").

csma. The first parameter of an instance refers to the constant N and the second parameter refers to the constant K. We consider the property

» $\mathsf{P}_{<\lambda}(\neg$"collision_max_backoff" U "all delivered").

firewire. The parameter of an instance refers to the constant *delay*. We consider the property

» $\mathsf{P}_{<\lambda}(\Diamond$ "done").

wlan. The first parameter of an instance refers to the constant *MAX_BACKOFF* and the second parameter refers to the constant *COL*. We consider the property

» $\mathsf{P}_{<\lambda}(\Diamond\, col = COL)$.

APPENDIX **D**

Symbolic Bisimulation Minimization of Markov Reward Automata

Models and Properties

Most models used for the evaluation in Chapter 5 were taken from the PRISM benchmark suite [KNP12] available at

`https://github.com/prismmodelchecker/prism-benchmarks/`.

Since this suite does not contain MA, we took these models from [QJK17] and [Guc+13] most of which can be found at

`https://github.com/moves-rwth/storm-examples`.

For every model, we match the parameters with (the names of) the constants of the model file and state the property in the syntax described in Appendix B.

bluetooth. This model is taken from the PRISM benchmark suite. The parameter of an instance refers to the constant *mrec*. We consider the property

» $\mathsf{R}_{=?}(\Diamond\ rec = mrec)$.

crowds. This model is taken from the Prism benchmark suite. The first parameter of an instance refers to the constant *CrowdSize* and the second parameter refers to the constant *TotalRuns*. We consider the property

» $\mathsf{P}_{=?}(\Diamond\ \mathit{observe0} > 1)$.

leader. This model is taken from the Prism benchmark suite. The first parameter of an instance refers to the constant *CrowdSize* and the second parameter refers to the constant *TotalRuns*. We consider the property

» $\mathsf{R}_{\text{num\_rounds}=?}(\Diamond\ \text{“elected”})$.

embedded. This model is taken from the Prism benchmark suite. The parameter of an instance refers to the constant *MAX_COUNT*. We consider the property

» $\mathsf{R}_{\text{down}=?}(\mathcal{C}^{\leq 168*3600})$.

polling. This model is taken from the Prism benchmark suite. The parameter of an instance refers to the constant N. We consider the property

» $\mathsf{R}_{\text{waiting}=?}(\mathcal{C}^{\leq 40})$.

p2p. This model is taken from the Prism website[1]. The first parameter of an instance refers to the constant N and the second parameter refers to the constant K. We consider the property

» $\mathsf{P}_{=?}(\Diamond^{\leq 1}\ \text{“done”})$.

coin. This model is taken from the Prism benchmark suite. The first parameter of an instance refers to the constant N and the second parameter refers to the constant K. We consider the property

» $\mathsf{R}^{+}_{\text{steps}=?}(\Diamond\ \text{“finished”})$.

[1] http://www.prismmodelchecker.org/casestudies/peer2peer.php

csma. This model is taken from the PRISM benchmark suite. The first parameter of an instance refers to the constant N and the second parameter refers to the constant K. We consider the property

» $\mathsf{R}^{+}_{\text{time}=?}(\lozenge \text{ “all\_delivered”})$.

wlan_dl. This model is taken from the PRISM benchmark suite. The first parameter of an instance refers to the constant *MAX_BACKOFF* and the second parameter refers to the constant *deadline*. We consider the property

» $\mathsf{P}^{-}_{=?}(\lozenge\ s1 = 12 \land s2 = 12)$.

mutex. This model is taken from [QJK17] and can be found in the repository located at `https://github.com/moves-rwth/storm-examples`. The parameter of an instance refers to the constant N. We consider the property

» $\mathsf{P}^{+}_{=?}(\lozenge^{\leq 5} \text{ “crit”})$.

polling. This model is taken from [Guc+13] and can be found in the repository located at `https://github.com/moves-rwth/storm-examples`. The first parameter of an instance refers to the constant N and the second parameter refers to the constant Q. We consider the property

» $\mathsf{P}^{-}_{=?}(\lozenge^{\leq 5} \text{ “q1full”})$.

jobs. This model is a slight variation of the model from [QJK17]. The original model can be found in the repository located at `https://github.com/moves-rwth/storm-examples`. Our variation concerns the number of different rates. In our model, only three different rates for the jobs are used. The first parameter of an instance refers to the constant N and the second parameter refers to the constant K. We consider the property

» $\mathsf{P}^{+}_{=?}(\lozenge^{\leq \frac{N}{4K}} \text{ “all\_jobs\_finished”})$.

Additional Data

In this section, we give additional experimental data. Table D.1 shows the number of nodes of the MTBDDs representing the transition matrices of the original model and the quotients. Furthermore, it give the number of iterations of the partition-refinement algorithm until it reached a fixed point.

Table D.2 gives the runtimes of symbolic versus (direct) sparse quotient extraction for all considered benchmark models.

	model	instance	$\lvert M_\Delta \rvert$	$\lvert M_{\Delta/\sim} \rvert$	iterations
DTMC	`bluetooth`	(1)	1.5×10^4	6.2×10^2	44
		(2)	1.8×10^6	1.6×10^6	307
	`crowds`	(30, 20)	3.2×10^5	3.4×10^3	138
		(30, 30)	5.0×10^5	3.6×10^3	208
	`leader`	(6, 8)	1.1×10^7	3.6×10^2	7
		(7, 7)	3.0×10^7	4.2×10^2	8
CTMC	`embedded`	(1000)	4.2×10^3	9.9×10^4	1004
		(2000)	4.6×10^3	1.2×10^5	2004
	`polling`	(16)	2.2×10^3	9.4×10^6	31
		(17)	2.5×10^3	2.0×10^7	33
	`p2p`	(7, 5)	5.5×10^4	1.1×10^4	34
		(8, 5)	6.9×10^4	1.3×10^4	39
PA	`coin`	(6, 4)	7.1×10^3	3.2×10^4	85
		(6, 6)	7.2×10^3	3.6×10^4	121
	`csma`	(3, 4)	7.5×10^4	1.1×10^5	42
		(4, 4)	5.5×10^5	1.5×10^6	48
	`wlan_dl`	(7, 140)	2.2×10^5	2.1×10^6	248
		(8, 140)	2.5×10^5	2.1×10^6	248
MA	`mutex`	(10)	1.5×10^4	4.0×10^5	10
		(15)	2.1×10^4	1.1×10^6	31
	`polling`	(3, 4)	2.9×10^3	2.8×10^5	10
		(4, 4)	3.5×10^3	1.8×10^6	10
	`jobs`	(15, 3)	2.9×10^6	3.7×10^6	12
		(16, 3)	6.7×10^6	6.8×10^6	12

Table D.1: Sizes of the quotient transient matrices and number of iterations.

	model	instance	symbolic	sparse
DTMC	bluetooth	(1)	0.65	0.18
		(2)	221.89	52.69
	crowds	(30, 20)	20.88	3.46
		(30, 30)	54.44	8.16
	leader	(6, 8)	41.75	8.44
		(7, 7)	137.04	49.93
CTMC	embedded	(1000)	13.01	1.11
		(2000)	46.24	2.44
	polling	(16)	488.26	17.16
		(17)	1398.98	38.46
	p2p	(7, 5)	48.39	0.38
		(8, 5)	71.93	0.57
PA	coin	(6, 4)	2.75	0.27
		(6, 6)	3.92	0.39
	csma	(3, 4)	12.01	2.29
		(4, 4)	409.41	66.12
	wlan_dl	(7, 140)	122.40	7.41
		(8, 140)	79.82	8.60
MA	mutex	(10)	31.63	4.40
		(15)	89.51	12.39
	polling	(3, 4)	12.55	2.51
		(4, 4)	136.12	25.54
	jobs	(15, 3)	180.05	29.71
		(16, 3)	423.10	87.85

Table D.2: Comparison of symbolic and sparse quotient extraction.

Appendix E

Game-Based Abstraction-Refinement

Most models used for the evaluation in Chapter 5 were taken from the Prism benchmark suite [KNP12] available at

`https://github.com/prismmodelchecker/prism-benchmarks/`.

The other models are taken from [Jan+16] and Pass' benchmark models, which are available at

`https://depend.cs.uni-saarland.de/tools/pass/casestudies/`.

For every model, we match the parameters with (the names of) the constants of the model file and state the property in the syntax described in Appendix B.

brp. This model is taken from the Prism benchmark suite. The first parameter of an instance refers to the constant N and the second parameter refers to the constant MAX. The third parameter identifies the property:

» p_1: $\mathsf{P}_{=?}(\lozenge\ s = 5)$,

» p_4: $\mathsf{P}_{=?}(\lozenge\ srep \neq 0 \wedge \neg recv)$.

coin. This model is taken from the Prism benchmark suite. The first parameter of an instance refers to the constant N and the second parameter refers to the constant K. The third parameter identifies the property:

» c_1: $\mathsf{P}_{\geq 1}(\Diamond \text{“finished”})$,

» c_2: $\mathsf{P}^{-}_{=?}(\Diamond \text{“finished”} \wedge \text{“all\_coins\_equal\_1”})$.

csma. This model is taken from Pass' benchmark suite and ultimately originates from Prism's benchmark suite. One of the modifications is setting the (internal) model constant λ to 50. The parameter of an instance refers to the constant K. The considered property is

» p_2: $\mathsf{P}^{-}_{=?}(\Diamond\ bc1 = 5 \vee bc2 = 5)$.

wlan. This model is taken from the Prism benchmark suite where we fix the value of the constant *COL* to 10. The first parameter of an instance refers to the constant *MAX_BACKOFF* and the second parameter identifies the property:

» min_3: $\mathsf{P}^{-}_{=?}(\Diamond\ bc1 = 3 \vee bc2 = 3)$,

» max_3: $\mathsf{P}^{+}_{=?}(\Diamond\ bc1 = 3 \vee bc2 = 3)$,

» max_6: $\mathsf{P}^{+}_{=?}(\Diamond\ bc1 = 6 \vee bc2 = 6)$.

zeroconf. This model is taken from the Prism benchmark suite where we fix the value of the constants N to 12000 and *reset* to *false*, respectively. The first parameter of an instance refers to the constant K and the second parameter identifies the property:

» min: $\mathsf{P}^{-}_{=?}(\Diamond\ l = 4 \wedge ip = 1)$,

» max: $\mathsf{P}^{+}_{=?}(\Diamond\ l = 4 \wedge ip = 1)$.

swp. This model is taken from Pass' benchmark suite. The parameter of the instance identifies the property:

» gp: $\mathsf{P}^{-}_{=?}(\Diamond\ sent > expected + 2)$,

» to: $\mathsf{P}^{+}_{=?}(\Diamond\ clock \geq 8)$.

coupon. This model is taken from [Jan+16] and was inspired by the famous coupon collector problem[1] from textbooks on randomized algorithms. The first parameter of an instance refers to the number of coupons to be collected and the second parameter refers to the number of coupons drawn in each round. The third parameter determines whether multiple identical coupons can be drawn within a round (r = regular) or whether the coupons are guaranteed to be distinct (c = conditional).

The **pGCL** code for two of the instances is given in Figures E.1 and E.2. We considered the property

» $\mathsf{P}_{=?}(\lozenge\ \mathit{numberDraws} > 5)$.

[1]`https://en.wikipedia.org/wiki/Coupon_collector%27s_problem`

```
function coupon() {
    var {
        bool coup0 := false;
        bool coup1 := false;
        bool coup2 := false;
        bool coup3 := false;
        bool coup4 := false;
        int draw0 := 0;
        int draw1 := 0;
        int draw2 := 0;
        int numberDraws := 0;
    }

    while (!coup0 | !coup1 | !coup2 | !coup3 | !coup4) {
        draw0 := unif(0, 4);
        draw1 := unif(0, 4);
        draw2 := unif(0, 4);
        numberDraws := numberDraws + 1;
        if (draw0 = 0 | draw1 = 0 | draw2 = 0) {
            coup0 := true;
        }
        if (draw0 = 1 | draw1 = 1 | draw2 = 1) {
            coup1 := true;
        }
        if (draw0 = 2 | draw1 = 2 | draw2 = 2) {
            coup2 := true;
        }
        if (draw0 = 3 | draw1 = 3 | draw2 = 3) {
            coup3 := true;
        }
        if (draw0 = 4 | draw1 = 4 | draw2 = 4) {
            coup4 := true;
        }
    }
}
```

Figure E.1: The **pGCL** code for instance `coupon(5,3,r)`.

```
function coupon() {
    var {
        bool coup0 := false;
        bool coup1 := false;
        bool coup2 := false;
        bool coup3 := false;
        bool coup4 := false;
        int draw0 := 0;
        int draw1 := 0;
        int draw2 := 0;
        int numberDraws := 0;
        bool first := true;
    }

    while (!coup0 | !coup1 | !coup2 | !coup3 | !coup4) {
        first := true;
        while (first | draw0 = draw1 | draw0 = draw2 | draw1 = draw2) {
            first := false;
            draw0 := unif(0, 4);
            draw1 := unif(0, 4);
            draw2 := unif(0, 4);
        }
        numberDraws := numberDraws + 1;
        if (draw0 = 0 | draw1 = 0 | draw2 = 0) {
            coup0 := true;
        }
        if (draw0 = 1 | draw1 = 1 | draw2 = 1) {
            coup1 := true;
        }
        if (draw0 = 2 | draw1 = 2 | draw2 = 2) {
            coup2 := true;
        }
        if (draw0 = 3 | draw1 = 3 | draw2 = 3) {
            coup3 := true;
        }
        if (draw0 = 4 | draw1 = 4 | draw2 = 4) {
            coup4 := true;
        }
    }
}
```

Figure E.2: The **pGCL** code for instance coupon(5,3,c).

Appendix F

Storm - A Modern Probabilistic Model Checker

All models used for the evaluation in Chapter 7 were taken from the Prism benchmark suite [KNP12] available at

https://github.com/prismmodelchecker/prism-benchmarks/.

Below, we list all considered models and and state the properties in the syntax described in Appendix B.

DTMCs

bluetooth. The parameter of an instance refers to the constant *mrec*. The considered property is:

» time: $\mathsf{R}_{=?}(\lozenge\ rec = mrec)$.

We considered the instances

» `bluetooth(0)` » `bluetooth(1)` » `bluetooth(2)`

brp. The first parameter of an instance refers to the constant N and the second parameter refers to the constant MAX. The third parameter identifies the property:

» p1: $\mathsf{P}_{=?}(\Diamond\ s = 5)$,

» p2: $\mathsf{P}_{=?}(\Diamond\ s = 5 \land \mathit{srep} = 2)$,

» p4: $\mathsf{P}_{=?}(\Diamond\ \mathit{srep} \neq 0 \land \neg \mathit{recv})$.

We considered the instances

» `brp(256,3,p1)`
» `brp(256,3,p2)`
» `brp(256,3,p1)`
» `brp(1024,4,p1)`
» `brp(1024,4,p2)`
» `brp(1024,4,p1)`
» `brp(4096,5,p1)`
» `brp(4096,5,p2)`
» `brp(4096,5,p1)`
» `brp(8192,6,p1)`
» `brp(8192,6,p2)`
» `brp(8192,6,p1)`

crowds. The first parameter of an instance refers to the constant *CrowdSize* and the second parameter refers to the constant *TotalRuns*. The considered property is:

» positive: $\mathsf{P}_{=?}(\Diamond\ \mathit{observe0} > 1)$.

We considered the instances

» `crowds(5,4)`
» `crowds(20,4)`
» `crowds(30,4)`
» `crowds(15,5)`
» `crowds(20,5)`
» `crowds(30,5)`
» `crowds(15,6)`
» `crowds(20,6)`
» `crowds(30,6)`
» `crowds(15,7)`
» `crowds(20,7)`
» `crowds(30,7)`

egl. The first parameter of an instance refers to the constant N and the second parameter refers to the constant L. The third parameter identifies the property:

» messagesA: $\mathsf{R}_{\text{messages\_A\_needs}=?}(\Diamond\ \mathit{phase} = 4)$,

» messagesB: $\mathsf{R}_{\text{messages\_B\_needs}=?}(\Diamond\ \mathit{phase} = 4)$,

» unfairA: $\mathsf{P}_{=?}(\Diamond\ \neg\text{“knowA”} \land \text{“knowB”})$,

» unfairB: $\mathsf{P}_{=?}(\Diamond\ \neg\text{“knowB”} \land \text{“knowA”})$.

We considered the instances

» `egl(5,8,messagesA)`
» `egl(5,8,messagesB)`
» `egl(5,8,unfairA)`
» `egl(5,8,unfairB)`
» `egl(10,8,messagesA)`
» `egl(10,8,messagesB)`
» `egl(10,8,unfairA)`
» `egl(10,8,unfairB)`
» `egl(20,8,messagesA)`
» `egl(20,8,messagesB)`
» `egl(20,8,unfairA)`
» `egl(20,8,unfairB)`

herman. The parameter of an instance refers to the number of processes. The considered property is:

» steps: $\mathsf{R}_{=?}(\lozenge \text{“stable”})$.

We considered the instances

» `herman(9)`
» `herman(11)`
» `herman(13)`
» `herman(15)`
» `herman(17)`
» `herman(19)`

leader. The first parameter of an instance refers to the constant *N* and the second parameter refers to the constant *K*. The third parameter identifies the property:

» eventually_elected: $\mathsf{P}_{\geq 1}(\lozenge \text{“elected”})$,
» time: $\mathsf{R}_{\text{num\_rounds}=?}(\lozenge \text{“elected”})$.

We considered the instances

» `leader(5,8,eventually_elected)`
» `leader(5,8,time)`
» `leader(6,5,eventually_elected)`
» `leader(6,5,time)`
» `leader(6,6,eventually_elected)`
» `leader(6,6,time)`

- `leader(6,8,eventually_elected)`
- `leader(6,8,time)`

nand. The first parameter of an instance refers to the constant N and the second parameter refers to the constant K. The considered property is:

- reliable: $\mathsf{P}_{=?}(\lozenge\ s = 4 \wedge z/N < 1/10)$.

We considered the instances

- `nand(20,1)`
- `nand(40,5)`
- `nand(40,7)`
- `nand(40,10)`
- `nand(60,5)`
- `nand(60,7)`
- `nand(60,10)`

CTMCs

cluster. The first parameter of an instance refers to the constant N and the second parameter refers to the constant T. The third parameter identifies the property:

- below_min: $\mathsf{R}_{\text{time\_not\_min}=?}(\mathcal{C}^{\leq T})$,
- operational: $\mathsf{R}_{\text{percent\_op}=?}(\mathcal{I}^{=T})$,
- premium_steady: $\mathsf{S}_{=?}(\text{“premium”})$,
- qos1: $\mathsf{P}_{=?}(\lozenge^{\leq T}\ \neg\text{“minimum”})$,
- qos2: $\mathsf{P}_{=?}(\lozenge^{=T}\ \neg\text{“minimum”})$,
- qos3: $\mathsf{P}_{=?}(\text{“minimum”}\ \mathsf{U}^{\leq T}\ \text{“premium”})$,
- qos4: $\mathsf{P}_{=?}(\neg\text{“minimum”}\mathsf{U}^{\geq T}\text{“minimum”})$,
- repairs: $\mathsf{R}_{\text{num\_repairs}=?}(\mathcal{C}^{\leq T})$.

We considered the instances

- `cluster(16,20,below_min)`
- `cluster(16,20,operational)`
- `cluster(16,20,premium_steady)`

- `cluster(16,20,qos1)`
- `cluster(16,20,qos2)`
- `cluster(16,20,qos3)`
- `cluster(16,20,qos4)`
- `cluster(16,20,repairs)`
- `cluster(256,20,below_min)`
- `cluster(256,20,operational)`
- `cluster(256,20,premium_steady)`
- `cluster(256,20,qos1)`
- `cluster(256,20,qos2)`
- `cluster(256,20,qos3)`
- `cluster(256,20,qos4)`
- `cluster(256,20,repairs)`
- `cluster(512,20,below_min)`
- `cluster(512,20,operational)`
- `cluster(512,20,premium_steady)`
- `cluster(512,20,qos1)`
- `cluster(512,20,qos2)`
- `cluster(512,20,qos3)`
- `cluster(512,20,qos4)`
- `cluster(512,20,repairs)`

embedded. The first parameter of an instance refers to the constant *MAX_COUNT* and the second parameter refers to the constant *T*. The third parameter identifies the property:

- actuators: $\mathsf{P}_{=?}(\neg\text{“down”}\ \mathsf{U}\ \text{“fail\_actuators”})$,
- actuators_T: $\mathsf{P}_{=?}(\neg\text{“down”}\ \mathsf{U}^{\leq T\cdot 3600}\ \text{“fail\_sensors”})$,
- danger_T: $\mathsf{R}_{\text{danger}=?}(\mathcal{C}^{\leq T\cdot 3600})$,

» danger_time: $\mathsf{R}_{\text{danger}=?}(\lozenge \text{ “down”})$,

» down_T: $\mathsf{R}_{\text{down}=?}(\mathcal{C}^{\leq T\cdot 3600})$,

» failure_T: $\mathsf{P}_{=?}(\lozenge^{\leq T\cdot 3600} \text{ “down”})$,

» io: $\mathsf{P}_{=?}(\neg\text{“down” } \mathsf{U} \text{ “fail\_io”})$,

» io_T: $\mathsf{P}_{=?}(\neg\text{“down” } \mathsf{U}^{\leq T\cdot 3600} \text{ “fail\_io”})$,

» main: $\mathsf{P}_{=?}(\neg\text{“down” } \mathsf{U} \text{ “fail\_main”})$,

» main_T: $\mathsf{P}_{=?}(\neg\text{“down” } \mathsf{U}^{\leq T\cdot 3600} \text{ “fail\_main”})$,

» sensors: $\mathsf{P}_{=?}(\neg\text{“down” } \mathsf{U} \text{ “fail\_sensors”})$,

» sensors_T: $\mathsf{P}_{=?}(\neg\text{“down” } \mathsf{U}^{\leq T\cdot 3600} \text{ “fail\_sensors”})$,

» up_T: $\mathsf{R}_{\text{up}=?}(\mathcal{C}^{\leq T\cdot 3600})$,

» up_time: $\mathsf{R}_{\text{up}=?}(\lozenge \text{ “down”})$.

The attentive reader might have noticed that the properties actuators_T and sensors_T are identical. We have contacted the authors of the benchmark suite to clarify whether this is by choice or an oversight. We considered the instances

» `embedded(2,20,actuators)`
» `embedded(2,20,actuators_T)`
» `embedded(2,20,danger_T)`
» `embedded(2,20,danger_time)`
» `embedded(2,20,down_T)`
» `embedded(2,20,failure_T)`
» `embedded(2,20,io)`
» `embedded(2,20,io_T)`
» `embedded(2,20,main)`
» `embedded(2,20,main_T)`
» `embedded(2,20,sensors)`
» `embedded(2,20,sensors_T)`
» `embedded(2,20,up_T)`
» `embedded(2,20,up_time)`
» `embedded(4,20,actuators)`
» `embedded(4,20,actuators_T)`
» `embedded(4,20,danger_T)`
» `embedded(4,20,danger_time)`
» `embedded(4,20,down_T)`
» `embedded(4,20,failure_T)`
» `embedded(4,20,io)`
» `embedded(4,20,io_T)`
» `embedded(4,20,main)`
» `embedded(4,20,main_T)`
» `embedded(4,20,sensors)`
» `embedded(4,20,sensors_T)`

- `embedded(4,20,up_T)`
- `embedded(4,20,up_time)`
- `embedded(7,20,actuators)`
- `embedded(7,20,actuators_T)`
- `embedded(7,20,danger_T)`
- `embedded(7,20,danger_time)`
- `embedded(7,20,down_T)`
- `embedded(7,20,failure_T)`
- `embedded(7,20,io)`
- `embedded(7,20,io_T)`
- `embedded(7,20,main)`
- `embedded(7,20,main_T)`
- `embedded(7,20,sensors)`
- `embedded(7,20,sensors_T)`
- `embedded(7,20,up_T)`
- `embedded(7,20,up_time)`
- `embedded(100,20,actuators)`
- `embedded(100,20,actuators_T)`
- `embedded(100,20,danger_T)`
- `embedded(100,20,danger_time)`
- `embedded(100,20,down_T)`
- `embedded(100,20,failure_T)`
- `embedded(100,20,io)`
- `embedded(100,20,io_T)`
- `embedded(100,20,main)`
- `embedded(100,20,main_T)`
- `embedded(100,20,sensors)`
- `embedded(100,20,sensors_T)`
- `embedded(100,20,up_T)`
- `embedded(100,20,up_time)`
- `embedded(1000,20,actuators)`
- `embedded(1000,20,actuators_T)`
- `embedded(1000,20,danger_T)`
- `embedded(1000,20,danger_time)`
- `embedded(1000,20,down_T)`
- `embedded(1000,20,failure_T)`
- `embedded(1000,20,io)`
- `embedded(1000,20,io_T)`
- `embedded(1000,20,main)`
- `embedded(1000,20,main_T)`
- `embedded(1000,20,sensors)`
- `embedded(1000,20,sensors_T)`
- `embedded(1000,20,up_T)`
- `embedded(1000,20,up_time)`

fms. The parameter of an instance refers to the constant N. The considered property is:

- productivity: $\mathsf{R}_{\text{productivity=?}}(\mathsf{S})$.

We considered the instances

- `fms(2)`
- `fms(4)`
- `fms(6)`
- `fms(8)`
- `fms(10)`

kanban. The parameter of an instance refers to the constant t. The considered property is:

- throughput: $\mathsf{R}_{\text{throughput}=?}(\mathsf{S})$.

We considered the instances

- `kanban(2)`
- `kanban(4)`
- `kanban(6)`
- `kanban(7)`

mapk_cascade. The first parameter of an instance refers to the constant N and the second parameter refers to the constant T. The third parameter identifies the property:

- activated_T: $\mathsf{R}_{\text{activated}=?}(\mathcal{I}^{=T})$,
- activated_time: $\mathsf{R}_{\text{time}=?}(\lozenge\ kpp = N)$,
- reactions: $\mathsf{R}_{\text{reactions}=?}(\mathcal{C}^{\leq T})$.

We considered the instances

- `mapk_cascade(4,20,activated_T)`
- `mapk_cascade(4,20,activated_time)`
- `mapk_cascade(4,20,reactions)`
- `mapk_cascade(6,20,activated_T)`
- `mapk_cascade(6,20,activated_time)`
- `mapk_cascade(6,20,reactions)`
- `mapk_cascade(8,20,activated_T)`
- `mapk_cascade(8,20,activated_time)`
- `mapk_cascade(8,20,reactions)`

polling. The first parameter of an instance refers to the constant N and the second parameter refers to the constant T. The third parameter identifies the property:

» s1: $\mathsf{S}_{=?}(s1 = 1 \wedge \neg(s = 1 \wedge a = 1))$,

» s1_before_s2: $\mathsf{P}_{=?}(\neg(s = 2 \wedge a = 1) \;\mathsf{U}\; (s = 1 \wedge a = 1))$,

» served: $\mathsf{R}_{\text{served}=?}(\mathcal{C}^{\leq T})$,

» station1_polled: $\mathsf{P}_{=?}(\lozenge^{\leq T}\; s = 1 \wedge a = 0)$,

» waiting: $\mathsf{R}_{\text{waiting}=?}(\mathcal{C}^{\leq T})$.

We considered the instances

» `polling(7,20,s1)`

» `polling(7,20,s1_before_s2)`

» `polling(7,20,served)`

» `polling(7,20,station1_polled)`

» `polling(7,20,waiting)`

» `polling(9,20,s1)`

» `polling(9,20,s1_before_s2)`

» `polling(9,20,served)`

» `polling(9,20,station1_polled)`

» `polling(9,20,waiting)`

» `polling(14,20,s1)`

» `polling(14,20,s1_before_s2)`

» `polling(14,20,served)`

» `polling(14,20,station1_polled)`

» `polling(14,20,waiting)`

» `polling(16,20,s1)`

» `polling(16,20,s1_before_s2)`

» `polling(16,20,served)`

» `polling(16,20,station1_polled)`

» `polling(16,20,waiting)`

tandem. The first parameter of an instance refers to the constant c and the second parameter refers to the constant T. The third parameter identifies the property:

» customers: $\mathsf{R}_{\text{customers}=?}(\mathsf{S})$,

» customers_T: $\mathsf{R}_{\text{customers}=?}(\mathcal{I}^{=T})$,

» first_queue: $\mathsf{P}_{=?}(\lozenge^{\leq T}\ sc = c)$,

» network: $\mathsf{P}_{=?}(\lozenge^{\leq T}\ sc = c \land sm = c \land ph = 2)$,

» second_queue: P=? [sm=c U<=T sm<c]; $\mathsf{P}_{=?}(sc = c\ \mathsf{U}^{\leq T}\ sm < c)$.

We considered the instances

» `tandem(15,0.2,customers)`

» `tandem(31,0.2,customers)`

» `tandem(511,0.2,customers)`

» `tandem(511,0.2,customers_T)`

» `tandem(511,0.2,first_queue)`

» `tandem(511,0.2,network)`

» `tandem(511,0.2,second_queue)`

» `tandem(1023,0.2,customers)`

» `tandem(1023,0.2,customers_T)`

» `tandem(1023,0.2,first_queue)`

» `tandem(1023,0.2,network)`

» `tandem(1023,0.2,second_queue)`

» `tandem(2047,0.2,customers)`

» `tandem(2047,0.2,customers_T)`

» `tandem(2047,0.2,first_queue)`

» `tandem(2047,0.2,network)`

» `tandem(2047,0.2,second_queue)`

PA

coin. The first parameter of an instance refers to the constant N and the second parameter refers to the constant K. The third parameter identifies the property:

» c1: $\mathsf{P}_{\geq 1}(\lozenge$ “finished”$)$,

» c2: $\mathsf{P}^{-}_{=?}(\lozenge$ “finished” $\land$ “all_coins_equal_1”$)$,

» disagree: $\mathsf{P}^{+}_{=?}(\lozenge$ “finished” $\land \neg$“agree”$)$,

» steps_max: $\mathsf{R}^{+}_{\text{steps}=?}(\lozenge$ “finished”$)$,

» steps_min: $\mathsf{R}^{-}_{\text{steps}=?}(\lozenge$ “finished”$)$.

We considered the instances

- » `coin(4,4,c1)`
- » `coin(4,4,c2)`
- » `coin(4,4,disagree)`
- » `coin(4,4,steps_max)`
- » `coin(4,4,steps_min)`
- » `coin(4,32,c1)`
- » `coin(4,32,c2)`
- » `coin(4,32,disagree)`
- » `coin(4,32,steps_max)`
- » `coin(4,32,steps_min)`
- » `coin(6,4,c1)`
- » `coin(6,4,c2)`
- » `coin(6,4,disagree)`
- » `coin(6,4,steps_max)`
- » `coin(6,4,steps_min)`
- » `coin(6,6,c1)`
- » `coin(6,6,c2)`
- » `coin(6,6,disagree)`
- » `coin(6,6,steps_max)`
- » `coin(6,6,steps_min)`

csma. The first parameter of an instance refers to the constant N and the second parameter refers to the constant K. The third parameter identifies the property:

- » all_before_max: $\mathsf{P}^{+}_{=?}(\neg\text{“collision\_max\_backoff”}\ \mathsf{U}\ \text{“all\_delivered”})$,
- » all_before_min: $\mathsf{P}^{-}_{=?}(\neg\text{“collision\_max\_backoff”}\ \mathsf{U}\ \text{“all\_delivered”})$,
- » some_before: $\mathsf{P}^{-}_{=?}(\lozenge\ \mathit{min\_backoff\_after\_success} < K)$,
- » time_max: $\mathsf{R}^{+}_{\text{time}=?}(\lozenge\ \text{“all\_delivered”})$,
- » time_min: $\mathsf{R}^{-}_{\text{time}=?}(\lozenge\ \text{“all\_delivered”})$.

We considered the instances

- » `csma(3,2,all_before_max)`
- » `csma(3,2,all_before_min)`
- » `csma(3,2,some_before)`
- » `csma(3,2,time_max)`
- » `csma(3,2,time_min)`
- » `csma(3,6,all_before_max)`
- » `csma(3,6,all_before_min)`
- » `csma(3,6,some_before)`
- » `csma(3,6,time_max)`
- » `csma(3,6,time_min)`
- » `csma(4,2,all_before_max)`
- » `csma(4,2,all_before_min)`
- » `csma(4,2,some_before)`
- » `csma(4,2,time_max)`

- `csma(4,2,time_min)`
- `csma(4,4,all_before_max)`
- `csma(4,4,all_before_min)`
- `csma(4,4,some_before)`
- `csma(4,4,time_max)`
- `csma(4,4,time_min)`

firewire. The first parameter of an instance refers to the constant *delay* and the second parameter refers identifies the property:

- elected: $\mathsf{P}_{\geq 1}(\Diamond \text{“done”})$,
- time_max: $\mathsf{R}^{+}_{\text{time}=?}(\Diamond \text{“done”})$,
- time_min: $\mathsf{R}^{-}_{\text{time}=?}(\Diamond \text{“done”})$,
- time_sending: $\mathsf{R}^{+}_{\text{time\_sending}=?}(\Diamond \text{“done”})$.

We considered the instances

- `firewire(36,elected)`
- `firewire(36,time_max)`
- `firewire(36,time_min)`
- `firewire(36,time_sending)`
- `firewire(100,elected)`
- `firewire(100,time_max)`
- `firewire(100,time_min)`
- `firewire(100,time_sending)`

firewire_abst. The first parameter of an instance refers to the constant *delay* and the second parameter refers identifies the property:

- elected: $\mathsf{P}_{\geq 1}(\Diamond \text{“done”})$,
- time_max: $\mathsf{R}^{+}_{\text{time}=?}(\Diamond \text{“done”})$,
- time_min: $\mathsf{R}^{-}_{\text{time}=?}(\Diamond \text{“done”})$,
- rounds: $\mathsf{R}^{-}_{\text{rounds}=?}(\Diamond \text{“done”})$.

We considered the instances

- `firewire_abst(36,elected)`
- `firewire_abst(36,time_max)`
- `firewire_abst(36,time_min)`
- `firewire_abst(36,rounds)`
- `firewire_abst(100,elected)`
- `firewire_abst(100,time_max)`

- `firewire_abst(100,time_min)`
- `firewire_abst(100,rounds)`

firewire_dl. The first parameter of an instance refers to the constant *delay* and the second parameter refers to the constant *deadline*. The considered property is:

- deadline: $\mathsf{P}^{-}_{=?}(\Diamond\ s = 9)$.

We considered the instances

- `firewire_dl(3,200)`
- `firewire_dl(3,800)`
- `firewire_dl(3,1000)`
- `firewire_dl(36,400)`
- `firewire_dl(36,800)`
- `firewire_dl(36,1000)`
- `firewire_dl(36,10000)`

firewire_impl_dl. The first parameter of an instance refers to the constant *delay* and the second parameter refers to the constant *deadline*. The considered property is:

- deadline: $\mathsf{P}^{-}_{=?}(\Diamond\ (s1 = 8 \wedge s2 = 7) \vee (s1 = 7 \wedge s2 = 8))$.

We considered the instances

- `firewire_impl_dl(36,1000)`
- `firewire_impl_dl(50,1000)`
- `firewire_impl_dl(100,1000)`

wlan. The first parameter of an instance refers to the constant *MAX_BACKOFF* and the second parameter refers to the constant *COL*. The third parameter identifies the property

- collisions: $\mathsf{P}^{+}_{=?}(\Diamond\ col = COL)$,
- cost_max: $\mathsf{R}^{+}_{\text{cost}=?}(\Diamond\ s1 = 12 \wedge s2 = 12)$,
- cost_min: $\mathsf{R}^{-}_{\text{cost}=?}(\Diamond\ s1 = 12 \wedge s2 = 12)$,
- num_collisions: $\mathsf{R}^{+}_{\text{collisions}=?}(\Diamond\ s1 = 12 \wedge s2 = 12)$,
- sent: $\mathsf{P}_{\geq 1}(\Diamond\ s1 = 12 \wedge s2 = 12)$,

» time_max: $\mathsf{R}^{+}_{\text{time}=?}(\lozenge\ s1 = 12 \land s2 = 12)$,

» time_min: $\mathsf{R}^{-}_{\text{time}=?}(\lozenge\ s1 = 12 \land s2 = 12)$.

We considered the instances

» `wlan(3,0,collisions)`
» `wlan(3,0,cost_max)`
» `wlan(3,0,cost_min)`
» `wlan(3,0,num_collisions)`
» `wlan(3,0,sent)`
» `wlan(3,0,time_max)`
» `wlan(3,0,time_min)`
» `wlan(5,0,collisions)`
» `wlan(5,0,cost_max)`
» `wlan(5,0,cost_min)`
» `wlan(5,0,num_collisions)`
» `wlan(5,0,sent)`
» `wlan(5,0,time_max)`
» `wlan(5,0,time_min)`
» `wlan(6,0,collisions)`
» `wlan(6,0,cost_max)`
» `wlan(6,0,cost_min)`
» `wlan(6,0,num_collisions)`
» `wlan(6,0,sent)`
» `wlan(6,0,time_max)`
» `wlan(6,0,time_min)`

wlan_dl. The first parameter of an instance refers to the constant *MAX_BACKOFF* and the second parameter refers to the constant *deadline*. The considered property is

» deadline: $\mathsf{P}^{-}_{=?}(\lozenge\ s1 = 12 \land s2 = 12)$.

We considered the instances

» `wlan(4,200,deadline)`
» `wlan(4,500,deadline)`
» `wlan(5,200,deadline)`
» `wlan(5,500,deadline)`
» `wlan(6,200,deadline)`
» `wlan(6,500,deadline)`

zeroconf. The first parameter of an instance refers to the constant N, the second parameter refers to the constant K and the third to the constant *reset*. The fourth parameter identifies the property:

» correct_max: $\mathsf{P}^{+}_{=?}(\lozenge\ l = 4 \land ip = 1)$,

» correct_min: $\mathsf{P}^{-}_{=?}(\lozenge\ l = 4 \wedge ip = 1)$.

We considered the instances

» `zeroconf(2000,2,false,correct_max)`

» `zeroconf(2000,2,false,correct_min)`

» `zeroconf(2000,2,true,correct_max)`

» `zeroconf(2000,2,true,correct_min)`

» `zeroconf(8000,8,false,correct_max)`

» `zeroconf(8000,8,false,correct_min)`

» `zeroconf(8000,8,true,correct_max)`

» `zeroconf(8000,8,true,correct_min)`

» `zeroconf(16000,16,false,correct_max)`

» `zeroconf(16000,16,false,correct_min)`

» `zeroconf(16000,16,true,correct_max)`

» `zeroconf(16000,16,true,correct_min)`

zeroconf_dl. The first parameter of an instance refers to the constant N, the second parameter refers to the constant K, the third parameter refers to the constant *reset* and the fourth parameter refers to the constant *deadline*. The fifth parameter identifies the property:

» deadline_max: $\mathsf{P}^{+}_{=?}(\neg(l = 4 \wedge ip = 2)\ \mathsf{U}\ t \geq \mathit{deadline})$,

» deadline_min: $\mathsf{P}^{-}_{=?}(\neg(l = 4 \wedge ip = 2)\ \mathsf{U}\ t \geq \mathit{deadline})$.

We considered the instances

» `zeroconf_dl(2000,2,false,50,deadline_max)`

» `zeroconf_dl(2000,2,false,50,deadline_min)`

» `zeroconf_dl(2000,2,true,50,deadline_max)`

» `zeroconf_dl(2000,2,true,50,deadline_min)`

- » `zeroconf_dl(8000,8,false,50,deadline_max)`
- » `zeroconf_dl(8000,8,false,50,deadline_min)`
- » `zeroconf_dl(8000,8,true,50,deadline_max)`
- » `zeroconf_dl(8000,8,true,50,deadline_min)`
- » `zeroconf_dl(16000,16,false,50,deadline_max)`
- » `zeroconf_dl(16000,16,false,50,deadline_min)`
- » `zeroconf_dl(16000,16,true,50,deadline_max)`
- » `zeroconf_dl(16000,16,true,50,deadline_min)`
- » `zeroconf_dl(16000,16,false,70,deadline_max)`
- » `zeroconf_dl(16000,16,false,70,deadline_min)`
- » `zeroconf_dl(16000,16,true,70,deadline_max)`
- » `zeroconf_dl(16000,16,true,70,deadline_min)`

2016-02	Ibtissem Ben Makhlouf: Comparative Evaluation and Improvement of Computational Approaches to Reachability Analysis of Linear Hybrid Systems
2016-03	Florian Frohn, Matthias Naaf, Jera Hensel, Marc Brockschmidt, and Jürgen Giesl: Lower Runtime Bounds for Integer Programs
2016-04	Jera Hensel, Jürgen Giesl, Florian Frohn, and Thomas Ströder: Proving Termination of Programs with Bitvector Arithmetic by Symbolic Execution
2016-05	Mathias Pelka, Grigori Goronzy, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 2nd KuVS Expert Talk on Localization
2016-06	Martin Henze, René Hummen, Roman Matzutt, Klaus Wehrle: The SensorCloud Protocol: Securely Outsourcing Sensor Data to the Cloud
2016-07	Sebastian Biallas: Verification of Programmable Logic Controller Code using Model Checking and Static Analysis
2016-08	Klaus Leppkes, Johannes Lotz, and Uwe Naumann: Derivative Code by Overloading in C++ (dco/c++): Introduction and Summary of Features
2016-09	Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, Peter Schneider-Kamp, and Cornelius Aschermann: Automatically Proving Termination and Memory Safety for Programs with Pointer Arithmetic
2016-10	Stefan Wüller, Ulrike Meyer, and Susanne Wetzel: Towards Privacy-Preserving Multi-Party Bartering
2017-01 *	Fachgruppe Informatik: Annual Report 2017
2017-02	Florian Frohn and Jürgen Giesl: Analyzing Runtime Complexity via Innermost Runtime Complexity
2017-04	Florian Frohn and Jürgen Giesl: Complexity Analysis for Java with AProVE
2017-05	Matthias Naaf, Florian Frohn, Marc Brockschmidt, Carsten Fuhs, and Jürgen Giesl: Complexity Analysis for Term Rewriting by Integer Transition Systems
2017-06	Oliver Kautz, Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe: CD2Alloy: A Translation of Class Diagrams to Alloy
2017-07	Klaus Leppkes, Johannes Lotz, Uwe Naumann, and Jacques du Toit: Meta Adjoint Programming in C++

2017-08	Thomas Gerlitz: Incremental Integration and Static Analysis of Model-Based Automotive Software Artifacts
2017-09	Muhammad Hamad Alizai, Jan Beutel, Jó Ágila Bitsch, Olaf Landsiedel, Luca Mottola, Przemyslaw Pawelczak, Klaus Wehrle, and Kasim Sinan Yildirim: Proc. IDEA League Doctoral School on Transiently Powered Computing
2018-01 *	Fachgruppe Informatik: Annual Report 2018
2018-02	Jens Deussen, Viktor Mosenkis, and Uwe Naumann: Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen
2018-03	Igor Kalkov: A Real-time Capable, Open-Source-based Platform for Off-the-Shelf Embedded Devices
2018-04	Andreas Ganser: Operation-Based Model Recommenders
2018-05	Matthias Terber: Real-World Deployment and Evaluation of Synchronous Programming in Reactive Embedded Systems

* These reports are only available as a printed version.
Please contact biblio@informatik.rwth-aachen.de to obtain copies.