Technische Universität München
TUM School of Computation, Information and Technology

Formal Verification of Algorithms for Automata and Model Checking

Julian M. Brunner

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz:

Prof. Dr. Helmut Seidl

Prüfende der Dissertation:

1. Prof. Tobias Nipkow, Ph.D.
2. Prof. Dr. Francisco Javier Esparza Estaun

Die Dissertation wurde am 25.10.2021 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 31.10.2022 angenommen.

Abstract

We use the proof assistant Isabelle to formally verify algorithms for omega automata and LTL model checking. The focus lies on generating executable code from these verified algorithms to produce highly trustworthy tools. These can then be used as reference implementations to test unverified tools against. Our work consists of three contributions in this general area.

Firstly, we formalize a library for transition systems and automata. It uses abstraction in order to support many different types of automata without duplicating formalization or compromising usability. The abstract part includes concepts like finite and infinite paths, language, degeneralization, and boolean operations on automata. Parts of it can also be instantiated on other structures like graphs and Petri nets. The concrete part includes several types of both finite and omega automata and can be extended easily.

Secondly, we formally verify an ample set partial order reduction algorithm. This includes the abstract correctness proof as well as the reduction algorithm which ensures that the reduction conditions hold. We integrate an executable implementation of this algorithm into the CAVA model checker. Furthermore, we show a counterexample for one of the lemmata involved in the proof of dynamic on-the-fly partial order reduction.

Thirdly, we formally verify algorithms for Büchi complementation and equivalence checking. The complementation algorithm follows the rank-based approach. We then use this algorithm in conjunction with both an intersection operation and an emptiness check to decide language-wise equivalence of Büchi automata. Finally, we integrate executable implementations of both of these algorithms into a command-line tool that can process automata represented in the Hanoi Omega-Automata format.

Zusammenfassung

Wir verwenden den Beweisassistenten Isabelle, um Algorithmen für Omega-Automaten und LTL Model Checking formal zu verifizieren. Der Schwerpunkt liegt auf der Generierung von ausführbarem Code, um vertrauenswürdige Werkzeuge zu produzieren. Diese können dann als Referenzimplementierungen verwendet werden, um unverifizierte Werkzeuge dagegen zu testen. Unsere Arbeit besteht aus drei Beiträgen in diesem Gebiet.

Erstens formalisieren wir eine Bibliothek für Transitionssysteme und Automaten. Sie verwendet Abstraktion, um viele verschiedene Typen von Automaten unterstützen zu können ohne Formalisierung zu duplizieren oder Einbußen bei der Benutzbarkeit zu machen. Der abstrakte Teil umfasst Konzepte wie endliche und unendliche Pfade, Sprache, Degeneralisierung, und boolesche Operationen auf Automaten. Teile davon können auch auf anderen Strukturen wie Graphen oder Petri-Netzen instanziiert werden. Der konkrete Teil umfasst mehrere Typen von sowohl endlichen als auch Omega-Automaten und kann leicht erweitert werden.

Zweitens verifizieren wir einen Ample Set Partial Order Reduction Algorithmus. Dies umfasst sowohl den abstrakten Korrektheitsbeweis als auch den Reduktionsalgorithmus, der die Reduktionsbedingungen sicherstellt. Wir integrieren eine ausführbare Implementierung dieses Algorithmus in den CAVA Model Checker. Weiterhin zeigen wir ein Gegenbeispiel für eines der Lemmata im Beweis von dynamischer on-the-fly Partial Order Reduction.

Drittens verifizieren wir Algorithmen für Büchi-Komplementierung und Äquivalenzprüfung. Der Komplementierungsalgorithmus folgt dem Rangbasierten Verfahren. Anschließend verwenden wir diesen Algorithmus zusammen mit einer Schnittoperation und einem Leerheitstest, um Sprachäquivalenz von Büchi-Automaten zu entscheiden. Schließlich integrieren wir ausführbare Implementierungen beider dieser Algorithmen in ein Kommandozeilenwerkzeug, welches im Hanoi Omega-Automata Format repräsentierte Automaten verarbeiten kann.

Acknowledgments

First and foremost, I want to thank my advisor Tobias Nipkow. Over the course of my undergraduate studies, he single-handedly redirected my interests from the field of software engineering towards the fields of logic and theoretical computer science via several lectures. Furthermore, his lecture *Semantics of Programming Languages* introduced me to Isabelle and the fascinating world of interactive theorem proving. Having been thoroughly hooked, I was both grateful and excited to be given the opportunity to become part of his research group as a doctoral student. During this time, his hands-off approach to supervision gave me almost complete freedom in how to pursue my research. At the same time, he was always available for advice and guidance, giving support and direction as needed.

I want to thank my colleagues at the *Chair for Logic and Verification* for making our workplace such a pleasant and friendly environment. During my time with the group, I met and worked with Mohammad Abdulaziz, Jasmin Blanchette, Manuel Eberl, Maximilian P. L. Haslbeck, Johannes Hölzl, Lars Hupel, Fabian Immler, Kevin Kappelmann, Ondřej Kunčar, Peter Lammich, Lars Noschinski, Andrei Popescu, Jonas Rädle, Simon Roßkopf, Lukas Stevens, Dmitriy Traytel, Simon Wimmer, and Bohua Zhan.

Among these people, I want to thank in particular Peter Lammich, who played an important role in many of the events that eventually brought me to the group. At the group, we worked together on the CAVA project. He is an expert in formalizing complex graph algorithms as well as deriving executable implementations for them. In this context, he also taught me the mysterious ways of the (in)famous refinement framework.

Next, I want to thank Ondřej Kunčar, who started the tradition of making movies for colleagues leaving the chair. I gladly served as the cinematographer for him and his fellow investigative journalist Simon Wimmer. Through their decisive improvisation, the movie was successfully produced in just a single afternoon and achieved widespread critical acclaim.

I also want to thank Manuel Eberl, who was always ready to include me in his shenanigans as well as take part in mine. This led to great success (or at least fun) in various endeavors like competitive Super Smash Bros. Melee for the Nintendo GameCube™, VRChat, a USB sledgehammer for Isabelle, a movie for Lars Hupel, and the Pub-Quiz.

Furthermore, I want to thank Mohammad Abdulaziz, another member of the Melee group. He was always full of friendly curiosity, willing to have conversations about Isabelle technicalities, research and work culture, society and politics, and even life advice.

Last but not least, I want to thank our office managers Eleni Nikolaou-Weiss and Helma Piller for helping us navigate the bureaucratic jungle that teaching and research at a university can be sometimes.

The research in this dissertation was supported by the DFG grants *Computer Aided Verification of Automata* (ES 139/5-1, NI 491/12-1, SM 73/2-1) and *Verified Model Checkers* (KR 4890/1-1, LA 3292/1-1). I want to thank all of the people taking part in these projects, namely Javier Esparza, Jan Křetínský, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, Salomon Sickert, Jan-Georg Smaus, and Thomas Tuerk. Without them, this research would not have been possible. I want to thank in particular my coauthors Peter Lammich, Salomon Sickert, and Benedikt Seidl. Finally, I want to thank Javier Esparza for providing his expertise on all things automata as well as his knowledge about the model-checking community.

There are several people who I have met or who have helped me during my research. These include Clemens Ballarin, Florian Haftmann, Alexander Krauss, Michael Luttenberger, Stephan Merz, Doron Peled, Albert Rizaldi, Helmut Seidl, Stephen Siegel, Antti Valmari, and Makarius Wenzel.

Finally, I want to thank Julian Asamer, Harald Brunner, Manuel Eberl, Yannick Mahlich, and Cameron Murri, who read preliminary versions of this dissertation and provided me with valuable feedback.

Contents

I Discussion **1**

1 Introduction **2**

2 Interactive Theorem Proving **5**
2.1 Isabelle 6
2.2 Verification 7
2.3 Refinement 8
2.4 Sequences 9

3 Automatic Verification **10**
3.1 Linear Temporal Logic 10
3.2 Omega Automata 11
3.3 Logic and Automata 11
3.4 Model Checking 12
3.5 Formalization 12

4 Transition Systems and Automata **14**
4.1 Architecture 15
4.2 Prerequisites 16
4.3 Transition Systems 17
4.4 Intermediates 20
4.5 Automata 22
4.6 Implementation 23
4.7 Formalization 24

5 Partial Order Reduction **25**
5.1 Theory 26
5.2 Abstract Correctness 27
5.3 Static Partial Order Reduction 28
5.4 Dynamic Partial Order Reduction 28

5.5 Results 31

6 Büchi Complementation **32**
6.1 Complementation 33
6.2 Equivalence 35
6.3 Results 35

7 Conclusion **37**

II Relevant Publications **41**

A Partial Order Reduction **42**
A.1 Introduction 45
A.2 Theory 46
A.2.1 Reduction Conditions 47
A.2.2 Reduction Algorithm 48
A.3 Formalization 49
A.3.1 Isabelle/HOL 49
A.3.2 Refinement Framework 50
A.3.3 Basics 50
A.3.4 Systems 50
A.3.5 Trace Theory 51
A.3.6 Abstract Correctness 52
A.3.7 The SM Language 55
A.3.8 Reduction Algorithm 56
A.3.9 Architecture of the CAVA Model Checker 56
A.3.10 Integration of Partial Order Reduction 59
A.4 Evaluation 60
A.5 Dynamic Partial Order Reduction with On-The-Fly Model Checking 60
A.6 Conclusion 62
References 63

B Büchi Complementation **65**
B.1 Introduction 67
B.2 Theory 68
B.2.1 Notation 69
B.2.2 Complementation 69
B.2.3 Complexity and Optimizations 70
B.2.4 Equivalence 71

B.3 Formalization 71
B.3.1 Isabelle/HOL 71
B.3.2 Basics 72
B.3.3 Transition Systems and Automata 72
B.3.4 Run DAGs 73
B.3.5 Odd Rankings 73
B.3.6 Complement Automaton 75
B.3.7 Refinement Framework 76
B.3.8 Implementation 77
B.3.9 Equivalence 78
B.3.10 Integration 78
B.4 Evaluation 79
B.5 Conclusion 81
References 81

III Extra Publications 85

C LTL Translation 86
C.1 Introduction 87
C.2 Preliminaries 90
C.3 Transition Systems and Automata 92
C.3.1 Abstract Transition Systems 93
C.3.2 Concrete Automata 94
C.3.3 Predefined Automata 96
C.3.4 Executable Implementation 96
C.3.5 Formalisation 96
C.3.6 Contributions to the Translation Formalisation . . . 96
C.4 The Master Theorem: Decomposing LTL Formulas 97
C.4.1 The “after”-Function 97
C.4.2 Syntactic Fragments of LTL 98
C.4.3 The Master Theorem 98
C.5 Deriving the DRA Construction 99
C.5.1 Constructing Automata for $L^1_{\varphi,X}$, $L^2_{X,Y}$, and $L^3_{X,Y}$ . . . 99
C.5.2 Assembling the Pieces 101
C.5.3 A Verified LTL Translator 101
C.6 Concluding Remarks 102
References 102

IV Bibliography 107

Part I

Discussion

Chapter 1

Introduction

Software is present in almost all parts of our lives, and so are its issues. They range from harmless quirks in smartphone user interfaces to critical failures in aviation control software. Given the ubiquity of these systems and the responsibility that they carry, avoiding these problems is essential.

Fundamentally, software issues constitute a discrepancy between what the software is supposed to do and what it actually does. To facilitate aligning these two sides, much of software engineering revolves around an intermediate concept called the specification. It is the result of a process called requirements engineering that gathers the requirements and refines them into a precise artifact. The actual software is then developed based on and according to this specification. In this view, the aforementioned discrepancies could be either due to the specification not properly capturing the requirements or the software not fulfilling the specification. Additionally, it is unfortunately not uncommon for there to be no explicit specification, with the developers left to guess what the software is meant to do. Over time, the discipline of software engineering has developed various ways of dealing with these issues. While the side of requirements engineering is interesting in its own right, it is not part of the topic of this dissertation. Instead, we will focus on correctness, the property of the software fulfilling its specification, as well as verification, the process of ensuring correctness.

Correctness can be approached from various angles and with varying levels of rigor. For instance, several practices and tools used during development can help bring the behavior of the software closer to its specification. These include but are not limited to software metrics, design patterns, best practices, and coding conventions. Assessing these indicators either manually

via code reviews or automatically via static analysis tools can help avoid incorrect behavior. A particularly important technique is testing, which can be done at various levels of granularity and during various stages in the development process. An approach called test-driven development is especially rigorous in that it encourages writing specifications for individual components in the form of tests. Thus, it incorporates both explicit specifications and testing into the development process itself, while also trying to take exhaustiveness into consideration via code coverage.

Depending on the software in question, a combination of these practices may be sufficient to achieve adequate levels of compliance with the specification. However, all of these techniques are by their nature incomplete and while they can help catch many mistakes, they can never guarantee correctness. In scenarios where this is required, we employ formal verification, the construction of a mathematical proof showing the correctness of an algorithm. Unfortunately, these proofs are typically verbose to the point of being impractically difficult to construct, maintain, understand, and review. Furthermore, a correctness proof that is at least as difficult to check as the correctness property itself does not seem very useful.

Fortunately, this is a great use case for interactive theorem proving. In this context, proofs are constructed with the help of and automatically checked for validity by a proof assistant. A proof assistant can automate some of the parts that make formal proofs tedious as well as abstract away details to make them less complex and easier to comprehend. This allows working with the proof at a higher and more abstract level while the full formal proof is still maintained in the background. Finally, the proof assistant ensures the validity of the proof at all times. This way, we can enjoy the benefits of formal verification while avoiding many of the drawbacks.

In order to use interactive theorem proving to perform formal verification, both the specification and the algorithm need to be stated in a formal and machine-readable language. This begs the question of whether it is possible to automatically check if the algorithm fulfills the specification, without a manually constructed correctness proof. Unfortunately, this problem is undecidable in general. That is, there is no algorithm that can compute the answer to this question for nontrivial specifications and arbitrary algorithms. Additionally, while constructing or guessing a proof can allow an algorithm to compute a positive answer in some cases, this is usually not viable in terms of algorithmic complexity. It is for this reason that proof assistants can only automate certain small steps of the proof, while the large-scale structure needs to be given by the user.

However, certain more restricted problems can indeed be automated. Notably, algorithms that describe finite systems can be exhaustively explored, giving rise to the idea of model checking. Here, a piece of software called a model checker decides whether a given algorithm fulfills a given specification. This takes place fully automatically, with the model checker either giving a positive answer or returning a counterexample.

Model checkers as well as their components and associated tools have become more and more complex over time. They incorporate a plethora of features, optimizations, and heuristics to improve usability and performance. Due to this, these verification tools have now themselves reached a point where their correctness can be called into question.

The correctness of verification tools is especially important since they act as trust multipliers. That is, the trust placed in these tools is extended to the systems that are checked by them. If they are not correct, this can easily lead to misplaced confidence in the correctness of these checked systems. In a sense, this could actually be more dangerous than not having used any verification techniques in the first place. Furthermore, verification tools are difficult to test on realistically-sized examples, as commonly another verification tool is the only way to verify the result. Finally, while the algorithms used in verification tools are sometimes examined by the scientific community, their implementations rarely are.

This leads to the main research topic of this dissertation. We want to use interactive theorem proving to formally verify algorithms associated with model checking. The immediate goal is to enable formally verified and thus highly trustworthy verification tools. Furthermore, these tools can then act as reference implementations to check other unverified tools against.

Chapters 2 and 3 will introduce the prerequisite concepts and literature in detail. Chapters 4, 5, and 6 cover the main research. Appendices C, A, and B contain the respective publications. Chapters 4 and 5 cover additional material not contained in their corresponding papers.

Chapter 2

Interactive Theorem Proving

In the broadest sense, interactive theorem proving describes the act of doing mathematics with the help of a computer. This is done with a piece of software called a proof assistant or theorem prover. The proof assistant allows expressing definitions, theorems, and proofs in a formal and machine-readable language. It then helps both in finding proofs and, perhaps more importantly, in checking their correctness.

As the name implies, the process of constructing a formal proof using such a system is interactive. Regular programming languages employ predictable type systems suitable for a loose edit-compile workflow. In contrast, formal proof construction requires more detailed and more immediate insight into the internal state of the prover. As such, it is both convenient and necessary to have a tight feedback loop between the user and the system.

Formal proofs could in theory also be done on paper. However, this is tedious, error-prone, and prohibitively time-consuming for all except the most trivial cases. As such, interactive theorem proving is not merely of academic interest, but enables the construction of formal proofs that would otherwise be infeasible. Formal proofs in turn ensure a level of rigor that would be difficult to achieve in an informal pen and paper proof.

Of course, one could argue that this merely shifts the question of correctness from the proof to the prover. It helps to think about this scenario in terms of the trusted codebase. It describes the minimum body of work that needs to be trusted in order to be able to trust the overall correctness theorem. In that sense, the use of a proof assistant means that instead of the proof itself, only the proof assistant needs to be trusted. If the proof assistant is small, this can reduce the trusted codebase significantly.

2.1 Isabelle

In this dissertation, we will focus mainly on the proof assistant Isabelle [NPW02, http://isabelle.in.tum.de]. It supports several logics, the most popular of which is Higher-Order Logic (HOL). It can be thought of as a combination of functional programming and logic.

Formalizations done in Isabelle are trustworthy due to its LCF-style architecture. In this architecture, there is a small logical inference kernel that all proofs have to pass through. Thus, the soundness of the entire system depends only on this inference kernel, with anything outside of the kernel unable to compromise soundness. This way, the trusted codebase is reduced to a minimum. Furthermore, the inference kernel is rarely modified and tested extensively over time. After decades of use and refinement, it constitutes the trustworthy foundation of the system.

One of the standout features of Isabelle is the Isar proof language, where Isar stands for *intelligible semi-automated reasoning*. It allows writing proofs in a way that resembles typical mathematical notation and which can be understood without information about the internal state of the prover. An example of such a proof is presented in the following.

Example 1 (Isar Proof)

theorem cantor:
  fixes f :: $a \Rightarrow a$ set
  shows $\neg$ surj f
proof
  assume surj f
  then obtain x **where** $f\ x = \{y.\ y \notin f\ y\}$ **by** force
  then have $x \in f\ x \longleftrightarrow x \in \{y.\ y \notin f\ y\}$ **by** blast
  also have $x \in \{y.\ y \notin f\ y\} \longleftrightarrow x \notin f\ x$ **by** blast
  finally have $x \in f\ x \longleftrightarrow x \notin f\ x$ **by** this
  then show False **by** simp
qed

Isabelle is supported by a rich standard library as well as the Archive of Formal Proofs (AFP) [https://www.isa-afp.org]. The standard library is part of the HOL object logic. It contains common mathematical concepts from fields such as order theory, algebra, analysis, number theory, combinatorics, and probability theory. It also includes concepts and data structures taken from functional programming as well as concepts related to semantics and formal verification. The AFP consists of various formal proof

developments mostly from the fields of mathematics and computer science. It covers a both wider and deeper range of topics than the standard library. It is continuously maintained by the authors of those entries as well as the Isabelle developers. The work in this dissertation both builds on some of these entries and contributes new ones to the archive.

2.2 Verification

While proof assistants allow for the expression of nearly arbitrary mathematics, we want to focus on using them to perform formal verification. To do so, both the specification and the algorithm have to be stated formally in the language of the proof assistant. In Isabelle, the algorithm is usually expressed using HOL definitions. The specification is a HOL proposition involving the constants arising from these definitions. This proposition can then be formally proven with the help of the proof assistant.

Note that while HOL definitions often resemble functional programs, they are not necessarily executable. They may contain arbitrary HOL terms, including uncomputable constructs like quantification over infinite sets. Even when they are executable, HOL is usually not the most suitable environment for software development. In this case, it may seem appealing to implement the verified algorithm in a separate programming language. However, by manually translating the algorithm, one risks introducing discrepancies. This way, the algorithm that ends up being executed may not be the same as the algorithm that was proven correct.

Isabelle includes a code generator that can export code from HOL theories to programming languages like SML, OCaml, Haskell, and Scala. To perform this, HOL constants are shallowly embedded in the target language and equational theorems are translated into rewrite rules [HN10]. Given the modest assumption that rewriting in the target language corresponds to equality in HOL, the exported constants have the same semantics as their HOL counterparts. Thus, the behavior should be the same and the correctness proof of the algorithm should transfer to its implementation.

It is worth noting that the code generator itself is not verified. However, this is still a significant improvement over having to establish and maintain the correspondence between algorithm and implementation manually. Overall, using interactive theorem proving in this way results in one of the strongest correctness guarantees possible for an algorithm.

2.3 Refinement

Formal verification often involves a trade-off between efficiency of the algorithm and simplicity of the proof. For instance, the use of efficient data structures can obscure the abstract and high-level ideas of the correctness proof with implementation details. Conversely, the representation most suitable for the proof is not necessarily the most efficient for execution. For complex algorithms, this can quickly become unmanageable.

Furthermore, many algorithms are naturally nondeterministic. For example, an algorithm may involve repeatedly picking and removing an element from a set until the set is empty. The order in which the elements are processed may not be relevant to the correctness of the algorithm. However, data structures that implement sets will determine a certain order in which its elements are picked. This mismatch can make both definitions and proofs very awkward, especially when induction is involved.

Both of these issues can be elegantly resolved via refinement [BW98]. In this approach, algorithms are initially expressed as abstract specifications, employing nondeterminism and abstract data representations. The abstract correctness proof of the algorithm is then performed on this specification. After that, the specification undergoes stepwise refinement towards an executable implementation. Each refinement step reduces nondeterminism by replacing parts of the specification with more concrete and specific versions. The structure and correctness of the algorithm are preserved in each step, such that each version inherits the abstract correctness proof.

The refinement approach enables separating the correctness proof of the algorithm from the correctness proof of the implementation. This way, the former is not encumbered by implementation details and the latter does not have to deal with the overall correctness of the algorithm. Furthermore, refinement is compositional, such that predefined data structures can be seamlessly substituted to implement abstract data representations.

Isabelle enables this via the Refinement Framework [Lam12; LT12; Lam16; Lam15]. It implements a refinement calculus [BW98] based on a nondeterminism monad [Wad92]. The modularity that comes with this approach in turn makes the Isabelle Collections Framework [Lam09; LL10] possible. It provides a library of verified and efficient data structures that can be reused in other formalizations. Together with powerful automation for tasks like verification condition generation and data structure substitution, these frameworks enable modular and efficient formal verification.

2.4 Sequences

Although extremely basic, sequences play an integral role in our work and thus deserve an explicit introduction. Fortunately, the HOL library already includes extensive support for both finite and infinite sequences in the form of the types list and stream.

Definition 1 (Sequences)

$$\textbf{datatype}\ \alpha\ \text{list} = []\ |\ \alpha\ \#\ \alpha\ \text{list} \tag{2.1}$$

$$\textbf{codatatype}\ \alpha\ \text{stream} = \alpha\ \#\#\ \alpha\ \text{stream} \tag{2.2}$$

The datatype list of finite sequences is well known from functional programming. The codatatype stream of infinite sequences is enabled by Isabelle's datatype package based on bounded natural functors [Bla+14; Bie+17].

It is worth noting that it is also possible to represent infinite sequences as functions $\mathbb{N} \Rightarrow \alpha$ that map each index to its corresponding item. This type is isomorphic to the type α stream. However, it can lead to undesired invocation of both higher-order unification and automation intended for proper functions. Furthermore, using a type synonym instead of a proper type prevents type class instantiation. Finally, the codatatype stream with its associated coinduction rules lends itself more easily to definitions and proofs involving corecursion and coinduction. These often allow expressing the desired concepts in a way that is more elegant and concise.

The libraries of both list and stream include many common operations and their properties, like the operators @ and @- for sequence concatenation.

Definition 2 (Concatenation)

$$\textbf{primrec}\ @ :: \alpha\ \text{list} \Rightarrow \alpha\ \text{list} \Rightarrow \alpha\ \text{list} \tag{2.3}$$

$$\textbf{primrec}\ @\text{-} :: \alpha\ \text{list} \Rightarrow \alpha\ \text{stream} \Rightarrow \alpha\ \text{stream} \tag{2.4}$$

We also added several definitions of our own relating to zipping, scanning, transposing, distinct sequences, and ordered sequences as well as some rules and automation for coinduction and infinite sequences.

In a more mathematical context, we will use the following notation. Instead of α list, we use Σ^*. Instead of α stream, we use Σ^ω. Let w^ω denote the infinite sequence resulting from infinite repetition of the finite sequence w. For a sequence $w \in \Sigma^*$ or $w \in \Sigma^\omega$, let $w_k \in \Sigma$ be the symbol at index k in w.

Chapter 3

Automatic Verification

Interactive theorem proving is a very powerful and general approach. Unfortunately, this generality comes at the price of requiring manual guidance. However, when it comes to verification, one is often interested in solving more constrained problems that can be automated to a higher degree.

One such approach is model checking [CGP01; Cla+18]. By restricting themselves to finite systems, model checkers can automatically decide if a given system satisfies a given specification. They do this by searching the state space of the system for a counterexample with respect to the given specification. As the state space is finite, they can search it exhaustively, thus either finding such a counterexample or concluding that there is none.

Popular model checkers include Spin [Hol04; Hol97] and PRISM [KNP11]. Another tool worth mentioning is the Alloy Analyzer [Jac06], which can also check infinite models, albeit not exhaustively.

In this dissertation, we will focus on the automata-theoretic approach to model checking [VW86] that is also used by Spin. The following sections will introduce the relevant concepts as well as how they relate to our work.

3.1 Linear Temporal Logic

Linear temporal logic (LTL) [Pnu77] is a type of modal logic that extends propositional logic with a temporal component. There are various ways of defining it that are equivalent in terms of expressive power. It usually contains the modal operators always, eventually, next, until, and release as

well as the operators from propositional logic. As we will treat LTL formulae opaquely in this dissertation, we will not define their interior structure.

We introduce some basic notation. LTL formulae are defined over a set of atomic propositions P. Let $\Sigma = 2^P$ denote the possible sets of atomic propositions. Each word $w \in \Sigma^\omega$ can then either satisfy a formula φ or not, denoted $w \models \varphi$. Let $\mathcal{L}\ \varphi \subseteq \Sigma^\omega$ denote the language of the LTL formula φ.

3.2 Omega Automata

Like finite automata, omega automata have a finite number of states. However, unlike finite automata, they operate on infinite sequences. Furthermore, omega automata come with a variety of acceptance conditions, like the Büchi, Rabin, and parity conditions. Some of these conditions also exist as both dualized and generalized versions.

Many omega automata recognize the same class of languages, the omega-regular languages. This class of languages generalizes the notion of regular languages and is closed under intersection, union, complementation, concatenation, and omega iteration. Throughout the rest of this dissertation, we will focus mostly on nondeterministic Büchi automata.

We introduce some basic notation. Let $A = (\Sigma, Q, I, \delta, F)$ be a nondeterministic Büchi automaton. It consists of an alphabet Σ, states Q, initial states $I \subseteq Q$, successor function $\delta :: \Sigma \Rightarrow Q \Rightarrow 2^Q$, and acceptance condition $F :: Q \Rightarrow \text{bool}$. Let $\mathcal{L}\ A \subseteq \Sigma^\omega$ denote the language of the automaton A.

3.3 Logic and Automata

An important property of LTL formulae is their ability to be translated to omega automata. A variety of algorithms and target automata are available for this [Ger+95; GO01; Bab+13a; Bab+13b; Sic+16; EKS16; EKS18]. Given an LTL formula φ, let A_φ be an automaton that recognizes the same language.

$$\mathcal{L}\ A_\varphi = \mathcal{L}\ \varphi \tag{3.1}$$

With this, it becomes appealing to model systems using omega automata and specifications using LTL formulae. The connection between these two structures then allows them to be processed by automata-theoretic algorithms. Because of this, LTL translation algorithms form the fundamental building blocks of many automatic verification techniques.

3.4 Model Checking

The basic idea of model checking is to automatically answer the question of whether a given system satisfies a given specification. When the system is given as an omega automaton S and the specification is given as an LTL formula φ, this problem can be stated in terms of their languages.

$$\mathcal{L}\, S \subseteq \mathcal{L}\, \varphi \tag{3.2}$$

As described in the previous section, LTL formulae can be translated to omega automata. This allows reducing the model checking problem to the emptiness problem on omega automata.

$$\begin{aligned} \mathcal{L}\, S \subseteq \mathcal{L}\, \varphi &\iff \mathcal{L}\, S \cap \overline{\mathcal{L}\, \varphi} = \emptyset && (3.3) \\ &\iff \mathcal{L}\, S \cap \mathcal{L}\, (\neg\varphi) = \emptyset && (3.4) \\ &\iff \mathcal{L}\, S \cap \mathcal{L}\, A_{\neg\varphi} = \emptyset && (3.5) \\ &\iff \mathcal{L}\, (S \times A_{\neg\varphi}) = \emptyset && (3.6) \end{aligned}$$

Step 3.3 is a simple set-theoretic transformation. Step 3.4 is based on negation of LTL formulae. Step 3.5 consists of the translation from LTL formulae to omega automata. Step 3.6 is a product construction on omega automata.

In the case of Büchi automata, emptiness can then be decided via algorithms for nested depth-first search [Cou+92; SE05] or strongly connected components [Gab00]. Thus, the whole process yields a decision procedure for the LTL model checking problem on Büchi automata.

3.5 Formalization

As mentioned before, both model checkers and associated automata algorithms and tools have become more and more complex over time. Furthermore, their role as trust multipliers means that their correctness is of particular importance. With this in mind, it seems appealing to verify the verification tools themselves, which brings us to the main topic of this dissertation. We want to use interactive theorem proving to create formally verified versions of these algorithms and tools.

Previous work in this area includes the DFG projects *Computer Aided Verification of Automata* and *Verified Model Checkers*. They are collectively referred to as the CAVA project [https://cava.in.tum.de]. Various components of model checkers have been formalized as part of the CAVA project [Lam14b;

Neu14; LN15; BL18; BSS19] as well as its preliminary work [SMS09]. These parts have then been integrated into a formally verified LTL model checker [Esp+14; Esp+13]. The AFP includes a formalization of linear temporal logic [Sic16]. Some of our work builds on and extends these formalizations.

The library for transition systems and automata described in chapter 4 forms the foundation of our work. Chapter 5 then describes how we use this library to extend the CAVA model checker with a partial order reduction optimization. Finally, we build a formally verified Büchi complementation and equivalence checking tool that is described in chapter 6.

Chapter 4

Transition Systems and Automata

Transition systems and automata are central concepts in theoretical computer science and especially formal language theory. They are also used in applications like regular expressions, model checking, and compiler construction. Given their importance and widespread use, it seems sensible to formalize them as a reusable library. Despite their shared name, automata cover a diverse range of structures, making this a challenging endeavor.

We start by establishing the goals of such a library.

Generality A general-purpose library should support many different types of automata and operations on them. This presents a significant challenge, as automata can vary independently along several dimensions. They can be deterministic or nondeterministic, each option including various representations. They can have labels on states or on transitions. They can employ a variety of acceptance conditions, including but not limited to finite acceptance, Büchi acceptance, and Rabin acceptance.

Abstraction The library should express concepts abstractly to facilitate sharing between different types of automata. This avoids duplication and enables more elegant and concise definitions and proofs. It also reduces the effort needed to extend the library with additional types of automata.

Usability The first two goals should not interfere with the usability of the library. Despite its generality and abstractness, it should offer good automation and require little overhead compared to a specialized library.

The Isabelle ecosystem already contains some ad-hoc automata formalizations as part of other work [Sic15; Esp+14; AVW16; LM07]. Several more have been developed as part of the CAVA project [LT12; Lam14a] and its

preliminary work [Mer00; SMS09]. However, all of these formalizations are either very specific or sacrifice usability to achieve their generality.

For instance, the CAVA automata library [Lam14a] supports only nondeterministic (generalized) Büchi automata with state labels and state-based acceptance. This makes it a perfect fit for applications like the CAVA model checker. However, its lack of generality makes it unsuitable for applications that require different types of automata or different representations.

A different formalization [LT12] models deterministic automata as a special case of nondeterministic ones. This seems appealing, as all instances of the former are also instances of the latter. However, it essentially bypasses the type system and requires manual management of wellformedness predicates in all proofs, greatly restricting automation. It also introduces a significant amount of boilerplate and glue code to account for the mismatch between types and values. Furthermore, this approach cannot support different representations of the same underlying concept.

4.1 Architecture

With this in mind, we propose the architecture shown in figure 4.1.

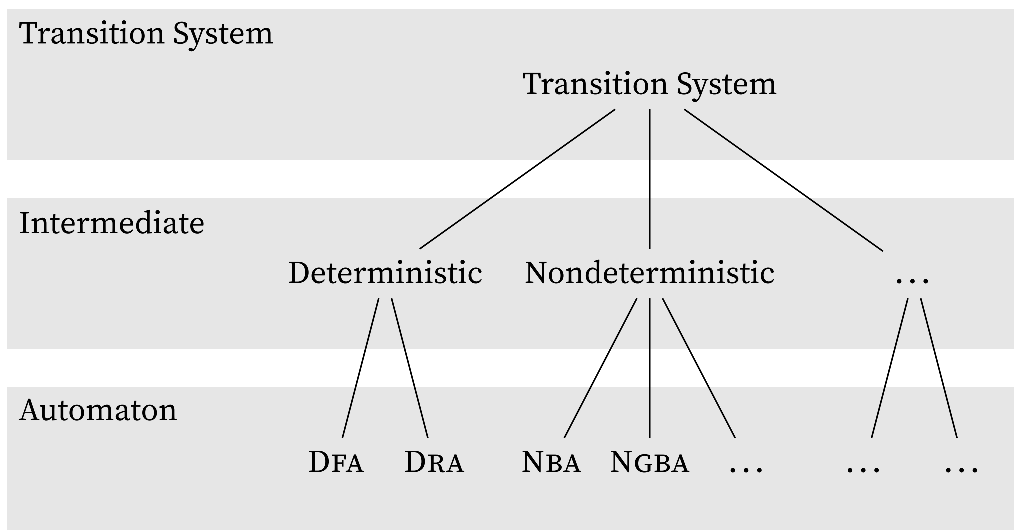


Figure 4.1: Architecture. The library consists of three layers that range from abstract and general at the top to concrete and specific at the bottom. Each item instantiates an item in the layer above. Not all items are shown. The ellipsis items indicate ways in which users can extend the library.

The central idea of this architecture is to formalize each concept on the most abstract layer possible. This maximizes sharing, enabling the items in the automaton layer to be expressed very concisely. To achieve this, we employ the locale mechanism provided by Isabelle [Bal14]. Using locales and instantiation rather than overly general types enables this degree of abstraction without compromising usability.

The system is as much a library as it is a set of tools for building custom automata formalizations. Depending on the situation, users can add new formalizations either on the intermediate layer or on the automaton layer. The former allows for maximum flexibility while still reusing the most abstract parts of the system. The latter enables adding new types of automata with minimal effort. With these options, users can represent automata in whichever way is most fitting for their application.

4.2 Prerequisites

Before describing the actual transition systems and automata library, there are some prerequisites that operate on the level of sequences.

In order to define omega acceptance conditions, we use a shallow embedding of linear temporal logic from the HOL library. It defines operations like alw and ev that express a property holding at every and any position of a sequence, respectively. With this, we can create a definition expressing that a predicate holds at infinitely many positions of a sequence.

Definition 3 (Infinite Occurrence)

$$\text{infs } P\ w \iff \text{alw (ev (holds } P))\ w \tag{4.1}$$

This is the Büchi acceptance condition. It is used to define others like the Rabin acceptance condition. We also define the higher-order combinators gen and cogen. They map acceptance conditions to their (co)generalized counterparts which act on sets of whatever acceptance component the original acceptance condition used.

Another important part of the library is degeneralization, the language-preserving translation from automata with (co)generalized (co-)Büchi acceptance conditions to regular (co-)Büchi automata. While it is difficult to share degeneralization as a whole between deterministic and nondeterministic automata, it turns out that core aspects of it can be formalized entirely at the level of sequences. This is done by defining constants for adding

acceptance class index annotations along an infinite sequence as well as a degeneralized acceptance test operating on these annotated sequences. An accompanying theorem then states that the annotated sequence is accepted by the regular acceptance condition if and only if the original sequence is accepted by the generalized acceptance condition. This formalization can then be used to define and prove correctness of degeneralization for any automaton with a suitable acceptance condition.

4.3 Transition Systems

The transition system layer forms the abstract root of the hierarchy. We model transition systems as objects that can use transitions to change state. This idea is directly reflected in their definition.

Definition 4 (Transition System)

$$\begin{aligned} &\textbf{locale}\ \text{transition_system} = \\ &\quad \textbf{fixes}\ \text{execute} :: \mathit{transition} \Rightarrow \mathit{state} \Rightarrow \mathit{state} \\ &\quad \textbf{fixes}\ \text{enabled} :: \mathit{transition} \Rightarrow \mathit{state} \Rightarrow \text{bool} \end{aligned} \tag{4.2}$$

This definition declares the types *transition* and *state* as well as the terms execute and enabled. The constant execute allows using a given transition to change state, while the constant enabled determines in which states a given transition can be executed. This very high level of abstraction allows the locale to be instantiated for many different structures that exhibit the characteristic of changing state via transitions.

While minimalistic, this already lets us define several basic constants.

Definition 5 (Targets and Traces)

$$\text{target} = \text{fold execute} :: \mathit{transition}\ \text{list} \Rightarrow \mathit{state} \Rightarrow \mathit{state} \tag{4.3}$$

$$\text{trace} = \text{scan execute} :: \mathit{transition}\ \text{list} \Rightarrow \mathit{state} \Rightarrow \mathit{state}\ \text{list} \tag{4.4}$$

$$\text{strace} = \text{sscan execute} :: \mathit{transition}\ \text{stream} \Rightarrow \mathit{state} \Rightarrow \mathit{state}\ \text{stream} \tag{4.5}$$

We employ the functions fold, scan, and sscan that are commonly used in functional programming. The constant target gives the target state of a finite sequence of transitions starting at a certain state. The constants trace and strace give the sequence of states traversed by a sequence of transitions starting at a certain state. Note how all of these constants are simply lifted versions of the constant execute, a fact that is also reflected in their types.

The transition system locale is also already sufficient to define the concepts of finite and infinite paths. We do so using (co)inductive predicates.

Definition 6 (Paths)

$$\textbf{inductive}\ \text{path} :: \textit{transition}\ \text{list} \Rightarrow \textit{state} \Rightarrow \text{bool}\ \textbf{where} \tag{4.6}$$
$$\text{path}\ []\ p$$
$$\text{enabled}\ a\ p \implies \text{path}\ r\ (\text{execute}\ a\ p) \implies \text{path}\ (a \mathbin{\#} r)\ p$$

$$\textbf{coinductive}\ \text{spath} :: \textit{transition}\ \text{stream} \Rightarrow \textit{state} \Rightarrow \text{bool}\ \textbf{where} \tag{4.7}$$
$$\text{enabled}\ a\ p \implies \text{spath}\ r\ (\text{execute}\ a\ p) \implies \text{spath}\ (a \mathbin{\#\#} r)\ p$$

The constants path and spath determine if a given sequence of transitions can be executed starting at a certain state. In analogy to definition 5, these constants are simply lifted versions of the constant enabled. In a similar way, we also supply inductive definitions for the set of reachable states from both a given state and the set of initial states.

While these definitions may seem trivial, they are fundamental to transition systems and automata and almost all other concepts depend on them. Because of this, even minor details can have a significant impact on usability. For instance, special care has been taken to define the constants path and spath in such a way that their application to composite sequences allows for safe introduction and elimination rules.

$$\text{path}\ (a \mathbin{\#} s)\ p \iff \text{enabled}\ a\ p \wedge \text{path}\ s\ (\text{execute}\ a\ p) \tag{4.8}$$
$$\text{path}\ (r \mathbin{@} s)\ p \iff \text{path}\ r\ p \wedge \text{path}\ s\ (\text{target}\ r\ p) \tag{4.9}$$
$$\text{spath}\ (a \mathbin{\#\#} s)\ p \iff \text{enabled}\ a\ p \wedge \text{spath}\ s\ (\text{execute}\ a\ p) \tag{4.10}$$
$$\text{spath}\ (r \mathbin{@\text{-}} s)\ p \iff \text{path}\ r\ p \wedge \text{spath}\ s\ (\text{target}\ r\ p) \tag{4.11}$$

For each kind of composite sequence, there is an equivalent statement on its components. Furthermore, each variable appears on both sides of its respective equation. This means that these rules can be safely applied to expand statements involving paths on composite sequences. This leads to significantly better automation than in other formalizations where these aspects have not been taken into account.

Next, we demonstrate how the transition system locale can be instantiated. For this, we consider some example transition systems with labels of type α and states of type σ. Each of those is represented by the function successor. We then give instantiations for the types and terms of the locale that enable accommodating this successor function.

Example 2 (Subdeterministic) successor :: $\alpha \Rightarrow \sigma \Rightarrow \sigma$ option

$$
\begin{aligned}
\mathit{transition} &= \alpha \\
\mathit{state} &= \sigma \\
\text{execute} &= \lambda\, a\, p.\ \text{the (successor } a\ p) && :: \alpha \Rightarrow \sigma \Rightarrow \sigma \\
\text{enabled} &= \lambda\, a\, p.\ \text{successor } a\ p \neq \text{None} && :: \alpha \Rightarrow \sigma \Rightarrow \text{bool} \\
\text{target} & && :: \alpha \text{ list} \Rightarrow \sigma \Rightarrow \sigma \\
\text{path} & && :: \alpha \text{ list} \Rightarrow \sigma \Rightarrow \text{bool}
\end{aligned}
$$

The deterministic successor function fits the interface straightforwardly. Furthermore, the constants target and path get the type signature that is expected for subdeterministic transition systems.

Example 3 (Nondeterministic) successor :: $\alpha \Rightarrow \sigma \Rightarrow \sigma$ set

$$
\begin{aligned}
\mathit{transition} &= \alpha \times \sigma \\
\mathit{state} &= \sigma \\
\text{execute} &= \lambda\, (a, q)\ p.\ q && :: \alpha \times \sigma \Rightarrow \sigma \Rightarrow \sigma \\
\text{enabled} &= \lambda\, (a, q)\ p.\ q \in \text{successor } a\ p && :: \alpha \times \sigma \Rightarrow \sigma \Rightarrow \text{bool} \\
\text{target} & && :: (\alpha \times \sigma) \text{ list} \Rightarrow \sigma \Rightarrow \sigma \\
\text{path} & && :: (\alpha \times \sigma) \text{ list} \Rightarrow \sigma \Rightarrow \text{bool}
\end{aligned}
$$

As mentioned before, it may seem appealing to model deterministic transition systems as a special case of nondeterministic ones. Furthermore, it may look like we are trying to do the impossible opposite here and represent a nondeterministic system as an instance of a deterministic one. However, it turns out that there is a surprisingly elegant solution. By instantiating the type variable *transition* with the type $\alpha \times \sigma$, there can be multiple transitions with the same label to different states. This turns out to be the right choice, in the sense that a value of type *transition* contains exactly the right amount of information. As the underlying locale treats values of type *transition* opaquely, this property is passed onto the constants defined earlier. For instance, the type of the constant path shown above contains a source state and a sequence of pairs of traversed labels and states. Thus, each value making up the path is mentioned exactly once and wellformedness is captured by the type directly.

By making use of locales and their ability to instantiate type variables rather than relying on overly general types, we achieve superior usability and flexibility. This enables sharing and reusing abstract parts of the formalization without incurring boilerplate or inconvenient wellformedness predicates for different types of transition systems. It also enables freely choosing

between isomorphic representations. For instance, we can use either an implicit successor function of type $\alpha \Rightarrow \sigma \Rightarrow \sigma$ set or an explicit transition set of type $(\sigma \times \alpha \times \sigma)$ set. These differences will then simply be absorbed by the locale instantiation.

Additionally, the transition system layer has been instantiated for several structures outside of the library for transition systems and automata. This includes systems with named operations [Bru18; BL18], directed acyclic graphs [Bru17a; Bru20], finite state systems [Sac19; Sac+19], and Petri nets. The ability of the formalization to accommodate these very different structures further demonstrates its generality.

4.4 Intermediates

The transition system layer contains the concepts that all automata and other instances have in common. Thus, it is very general, but also very basic, capturing only the very core of what an automaton is. There are many things that some but not all automata share and which simply require more specificity than the transition system layer provides. For instance, defining the union of two automata may be almost identical among all nondeterministic automata, but work very differently for deterministic ones. Furthermore, the transition system layer provides a read-only view of a single instance. Due to this, it does not support multiple automata or the ability to assemble a new automaton from its components.

To solve these issues, we introduce an intermediate layer into the hierarchy. We present the nondeterministic branch of this layer as an example.

Definition 7 (Nondeterministic Automaton)

$$\begin{aligned}
&\textbf{locale}\ \text{automaton_nondeterministic} = \\
&\quad \textbf{fixes}\ \text{automaton} :: \textit{label}\ \text{set} \Rightarrow \textit{state}\ \text{set} \Rightarrow \\
&\qquad (\textit{label} \Rightarrow \textit{state} \Rightarrow \textit{state}\ \text{set}) \Rightarrow \textit{acceptance} \Rightarrow \textit{automaton} \\
&\quad \textbf{fixes}\ \text{alphabet} :: \textit{automaton} \Rightarrow \textit{label}\ \text{set} \\
&\quad \textbf{fixes}\ \text{initial} :: \textit{automaton} \Rightarrow \textit{state}\ \text{set} \\
&\quad \textbf{fixes}\ \text{transition} :: \textit{automaton} \Rightarrow (\textit{label} \Rightarrow \textit{state} \Rightarrow \textit{state}\ \text{set}) \\
&\quad \textbf{fixes}\ \text{acceptance} :: \textit{automaton} \Rightarrow \textit{acceptance}
\end{aligned} \tag{4.12}$$

This locale differs from the one for transition systems from definition 4 in several ways. Firstly, it includes the automaton as an explicit term with a constructor and selectors. The constructor allows assembling automata

from parts and the selectors allow expressing statements that involve more than one automaton, neither of which is possible in the transition system locale. Also note how the type of the acceptance component is simply a type variable and thus has no structure. This way, the locale makes no assumptions about the acceptance component and is thus completely independent of the acceptance condition that is used. Finally, this being the intermediate layer, the transition component takes on a more specific type. This allows defining operations like intersection and union that depend on the type of the transition component.

In order to express concepts like the language of an automaton, we need to be able to extract information from the acceptance component. To this end, we extend the locale from definition 7 with additional terms for finite and infinite path acceptance, respectively.

Definition 8 (Nondeterministic Automaton with Acceptance)

locale automaton_nondeterministic_finite = (4.13)
automaton_nondeterministic +
fixes test :: *acceptance* ⇒ *label* list ⇒ *state* list ⇒ *state* ⇒ bool

locale automaton_nondeterministic_infinite = (4.14)
automaton_nondeterministic +
fixes test :: *acceptance* ⇒ *label* stream ⇒ *state* stream ⇒ *state* ⇒ bool

The term test represents the acceptance condition. With this setup, only the decision of whether the automaton operates on finite or infinite sequences has to be made on the intermediate layer. In fact, the term test is even general enough to accommodate both state-based and transition-based acceptance. Meanwhile, the specific acceptance condition used by the automaton is injected into the locale only at the time of instantiation.

The desired flexibility of the library with respect to aspects like finiteness, determinism, acceptance, and representation has a multiplicative effect on the number of types of automata. This can quickly become overwhelming, such that even small amounts of required boilerplate code can have large effects. Due to this, sharing of formalization is not a mere convenience, but can actually determine whether incorporating certain concepts uniformly and for all types of automata is feasible at all. The intermediate layer allows for expressing and sharing of concepts such as language, degeneralization, and boolean operations on automata. This way, it becomes significantly easier to add new types of automata to the formalization.

4.5 Automata

Finally, the automaton layer forms the leaves of the hierarchy. In this layer, we define concrete datatypes for specific automata and use them to instantiate the locales that make up the more abstract layers.

We present this procedure for the example of nondeterministic finite automata. The first step consists of defining a datatype for the automaton.

Definition 9 (Nondeterministic Finite Automaton)

$$\begin{aligned}&\textbf{datatype}\ (\mathit{label}, \mathit{state})\ \mathrm{nfa} = \mathrm{nfa}\\ &\quad (\mathrm{alphabet} : \mathit{label}\ \mathrm{set})\\ &\quad (\mathrm{initial} : \mathit{state}\ \mathrm{set})\\ &\quad (\mathrm{transition} : \mathit{label} \Rightarrow \mathit{state} \Rightarrow \mathit{state}\ \mathrm{set})\\ &\quad (\mathrm{accepting} : \mathit{state} \Rightarrow \mathrm{bool})\end{aligned} \tag{4.15}$$

We can then instantiate the locales from the intermediate layer. In this example, the locale for finite path acceptance from definition 8 is used.

$$\begin{aligned}&\textbf{interpretation}\ \mathrm{nfa} : \mathrm{automaton_nondeterministic_finite}\\ &\quad \mathrm{nfa\ alphabet\ initial\ transition\ accepting}\\ &\quad \lambda\, P\, w\, r\, p.\ P\ (\mathrm{last}\ (p \# r))\end{aligned} \tag{4.16}$$

This immediately gives us access to definitions and theorems for all of the concepts from the transition system and intermediate layers. For our finite automaton example, this includes the following definitions among others.

$$\mathrm{path} :: (\mathit{label}, \mathit{state})\ \mathrm{nfa} \Rightarrow (\mathit{label} \times \mathit{state})\ \mathrm{list} \Rightarrow \mathit{state} \Rightarrow \mathrm{bool} \tag{4.17}$$

$$\mathrm{nodes} :: (\mathit{label}, \mathit{state})\ \mathrm{nfa} \Rightarrow \mathit{state}\ \mathrm{set} \tag{4.18}$$

$$\mathrm{language} :: (\mathit{label}, \mathit{state})\ \mathrm{nfa} \Rightarrow \mathit{label}\ \mathrm{list\ set} \tag{4.19}$$

Note how the types of these constants are exactly what one would expect from a tailored formalization of nondeterministic finite automata. The fact that they are instantiated from a more abstract structure is thus barely visible and largely does not impact usability on the concrete level.

Furthermore, adding degeneralization, intersection and union operations is just a matter of instantiating another locale for each. Overall, thanks to sharing most of the formalization, the entire setup necessary to get a reasonable automaton instance involves less than 50 lines of theory text.

In the case of Büchi automata, the situation is slightly more complicated as their intersection operation involves degeneralization [Var95; Cho74].

Specifically, the intersection of two Büchi automata results in a generalized Büchi automaton, which is subsequently degeneralized to a regular Büchi automaton. To enable this process, we first set up formalizations for both regular and generalized Büchi automata. Next, the intermediate layer can be instantiated to provide the necessary intersection and degeneralization operations. These operations can then be composed to form an intersection operation which acts entirely on regular Büchi automata. This compositional approach cleanly separates Büchi intersection into its component operations and makes use of the generality of the intermediate layer. While more complicated than finite automata, a reasonable formalization of Büchi automata including intersection via generalized Büchi automata can still be achieved in less than 100 lines.

The library comes with several concrete automaton types. On the deterministic side, these include finite automata, (generalized) (co-)Büchi automata, and Rabin automata. On the nondeterministic side, these include finite automata and (generalized) Büchi automata. Both the deterministic and the nondeterministic Büchi automata support both state-based and transition-based acceptance. More automaton types can be easily added as needed.

As mentioned before, the main focus of the library is not on providing a comprehensive collection of concrete automata formalizations. Experience has shown that many use cases have very specific requirements when it comes to automata. Thus, our main goal is to provide the tools necessary to assemble custom automata formalizations with minimal effort. This enables each user to work with the representation that is most suitable to their needs, while reusing as much of the existing formalization as possible.

4.6 Implementation

In addition to formalizing abstract automata theory, we also want to use our library to create formally verified and executable algorithms. Furthermore, many of the algorithms acting on automata involve nondeterminism and thus require heavy machinery like the refinement framework introduced in section 2.3. Due to this, a certain amount of setup is needed, including the definition of relations between abstract automata and their implementations. With this setup, we can use automatic refinement [Lam13a; Lam13b] to generate executable implementations for our definitions. Finally, these executable implementations can then be exported using Isabelle's code generation facilities that were introduced in section 2.2.

While some definitions like degeneralization and intersection are fairly basic and can be implemented mostly automatically, others form complex algorithms. For instance, enumerating the set of reachable states in an automaton is done via depth-first search. For this, we use the depth-first search framework [LN16; LN15] that is available in the AFP.

This algorithm can then be used to turn an automaton that is implicitly defined by a set of initial states and a transition function into an explicit representation like a transition list. Furthermore, it allows assigning a unique index to each state. We use this to implement a language-preserving translation from arbitrary state types to natural numbers.

Unfortunately, implementations for algorithms like these are naturally very specific to each type of automaton and thus difficult to share. Due to this, we chose to formalize executable implementations separately for each automaton type as needed.

4.7 Formalization

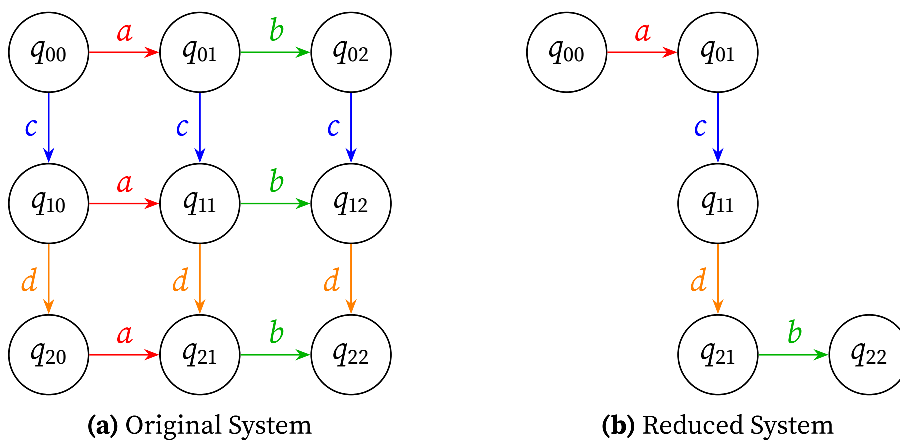
The library for transition systems and automata is available as part of the AFP [Bru17b]. It has been used in several other formalizations [Bru18; BL18; SS19; BSS19; Sac19; Sac+19; Bru17a; Bru20].

In particular, the library enabled the formalization of the compositional LTL formula to automata translation procedure described in [BSS19]. This approach makes use of several different types of automata as well as inhomogenous boolean operations. For instance, it requires an intersection operation between a deterministic Büchi automaton and a deterministic co-Büchi automaton resulting in a deterministic Rabin automaton. Due to sharing in the intermediate layer, even specialized constructions like this one do not incur any duplication of either definitions or proofs.

Chapter 5

Partial Order Reduction

When modeling the behavior of concurrent systems, all possible interleavings of operations in the participating processes have to be considered. In the context of model checking, this can quickly become unmanageable and lead to what is known as state space explosion. To counteract this, several partial order reduction techniques have been proposed [Val89; God90; God96; Pel93; Pel98]. These algorithms identify situations in which the order of operations is not important and only a subset of all interleavings has to be examined. Figure 5.1 shows an example of this. These techniques can dramatically reduce the state space and thus improve performance.



(a) Original System

(b) Reduced System

Figure 5.1: Partial Order Reduction Example. The operations a and b are independent of the operations c and d. This means that they can be executed in any order without changing the result. Thus, only one interleaving has to be considered for properties that are not sensitive to this order.

Partial order reduction algorithms are complex and have proven to be notoriously difficult to get right. For instance, the ample set algorithm in [Pel96] was believed to be compatible with nested depth-first search. However, it later turned out that the algorithm could choose different reductions in the outer and the inner search, leading to incorrect behavior [HPY96]. During our formalization efforts of [Pel96], we found additional problems with the proof [Bru14; BL18]. These were later found out to also cause incorrect behavior of the algorithm and affect results of the Spin model checker [Sie19]. Finally, another issue was discovered in the stubborn set partial order reduction algorithm [NVW20]. Here, the conditions are not strong enough and allow for cases that produce an incorrect result. However, all implementations of the algorithm happen to work with an approximation of these conditions that cannot represent these cases.

Given this historical and current context, it appears desirable to formally verify these algorithms. For this, we focus on the ample set flavor of partial order reduction [Pel93; Pel96; Pel98]. A version of this technique was already independently formalized in a different proof assistant [CP96]. However, it only covers the abstract part, requires a specific fairness assumption, and is not compatible with on-the-fly model checking.

Our original goal was to formalize dynamic on-the-fly partial order reduction [Pel96]. This means that the algorithm takes dynamic system state into account when making decisions about the reduction. It also means that the reduction takes place on-the-fly, as part of the emptiness check in the model checking process, without explicitly constructing the entire reduced state space. Unfortunately, due to the aforementioned issues that will be explained in detail in section 5.4, this was not possible. Instead, we chose to formalize static partial order reduction [Kur+98], which bases its decisions on approximations derived from static control-flow analysis.

5.1 Theory

On an abstract level, partial order reduction takes a system S and derives a reduced system R from it. This reduced system is then used for model checking instead of the original system. While $\mathcal{L}\, S$ and $\mathcal{L}\, R$ are not necessarily equal, the reduction algorithm ensures that the property φ cannot distinguish between them. Thus, model checking with the reduced system yields the same result as model checking with the original system.

$$\mathcal{L}\, S \subseteq \mathcal{L}\, \varphi \iff \mathcal{L}\, R \subseteq \mathcal{L}\, \varphi \tag{5.1}$$

The theory of ample set partial order reduction is centered around the concept of reduction conditions. These describe a set of properties that are sufficient for the reduction to not affect the result of the model checker. This approach neatly splits the correctness proof into two parts, decoupling reasoning about the actual reduction construction from the complex abstract correctness proof of partial order reduction.

We will only introduce one of these reduction conditions here. Condition *well-founded* requires that every cycle in the reduced system contains at least one state where no reduction is performed. This ensures that the execution of independent operations is not deferred indefinitely. For a detailed explanation of the other reduction conditions, see appendix A.

5.2 Abstract Correctness

We build on our formalization of transition systems and automata from chapter 4 to define systems with operations as required by ample set partial order reduction. We also formalize parts of trace theory [Maz86], which lifts the idea of independent operations to that of equivalent sequences of operations. Furthermore, ample set partial order reduction involves sequences which are not known to be either finite or infinite. To accommodate these sequences, we use a formalization of lazy lists [Loc10] from the AFP and extend it with limits and index sets. Finally, we use a formalization of stuttering-equivalence of infinite sequences [Mer12] from the AFP.

With these tools, we can formalize the abstract correctness proof of ample set partial order reduction. We assume that the reduced system R meets the reduction conditions and formally prove lemma 3.7 as well as theorems 3.9, 3.11, and 3.12 from [Pel96]. This yields the following statement.

Theorem 1 (Stuttering Equivalence)

$$w \in \mathcal{L}\, R \implies w \in \mathcal{L}\, S \tag{5.2}$$

$$u \in \mathcal{L}\, S \implies \exists\, v \in \mathcal{L}\, R.\ u \approx v \tag{5.3}$$

It states that the language of the reduced system is a subset of the language of the original system. Furthermore, it states that for each word accepted by the original system, there exists a word that is equivalent up to stuttering that is also accepted by the reduced system. Together with the fact that properties expressed by next-free LTL formulae are stuttering-invariant [PW97], we get the final correctness theorem.

Theorem 2 (Correctness of Ample Set Partial Order Reduction)

$$\text{next_free}\ \varphi \implies (\mathcal{L}\ S \subseteq \mathcal{L}\ \varphi \iff \mathcal{L}\ R \subseteq \mathcal{L}\ \varphi) \tag{5.4}$$

5.3 Static Partial Order Reduction

While the previous section was concerned with the abstract correctness proof, we now turn to the issue of actually constructing the reduced system in such a way that the reduction conditions hold. For instance, condition well-founded states that reductions cannot take place at every state in a cycle of the reduced system. Dynamic partial order reduction ensures this condition with respect to the actual state space of the reduced system. In contrast, static partial order reduction [Kur+98] overapproximates it by ensuring that it holds on the control-flow graph of the reduced system. It does this by identifying a set of so-called sticky edges that break every cycle in the control-flow graph and then preventing any reduction involving these edges. Thus, it does not base reduction decisions on information available at run-time, relying on results of a static analysis phase instead.

While this more conservative approach may cause less reduction overall, it has significant advantages. Firstly, it avoids all of the issues covered in section 5.4. Secondly, and perhaps not any less importantly, it drastically increases modularity. While dynamic on-the-fly partial order reduction collapses reduction, product construction, and emptiness check into one monolithic algorithm, static partial order reduction allows for a compositional approach. This makes it much more manageable, especially in the context of formal verification.

5.4 Dynamic Partial Order Reduction

As mentioned previously, dynamic partial order reduction ensures condition well-founded dynamically, by inspecting the search stack during the depth-first search exploration of the system. This way, it can detect which transitions would close a cycle in the state space of the system and prevent reduction from taking place in these situations. In the case of on-the-fly model checking, this exploration takes place as part of the emptiness check on the product automaton. This requires some adjustments to both the partial order reduction algorithm and the model checker.

One such approach is presented in [Pel96]. Here, partial order reduction, product automaton, and emptiness check all collapse into one monolithic

algorithm, requiring a specialized construction. Let S be the system with interpretation $\mathcal{I}$ and let A_φ be the automaton for the formula φ. Then the successor function of the reduced product $S \times A_\varphi$ is defined as follows.

Definition 10 (Reduced Product Successor)

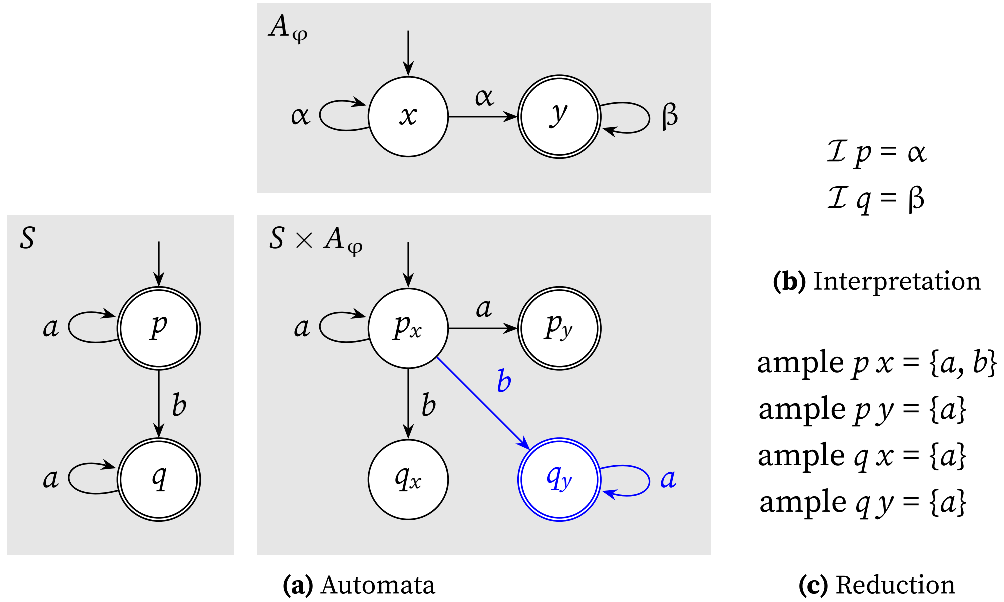
$$q_y \in \delta_{S\times A_\varphi}\; a\; p_x \iff y \in \delta_{A_\varphi}\; (\mathcal{I}\; p)\; x \wedge a \in \text{ample}\; p\; y \wedge q \in \delta_S\; a\; p \qquad (5.5)$$

Here, a transition takes place in two steps. Since the formula automaton A_φ can exhibit nondeterminism, it needs to perform its transition first. This way, the reduction function ample can get access to both the current system state p and the formula successor state y. Together with the depth-first search stack, it can furthermore determine if a cycle in the state space of the system would be closed by the transition. It can then use this information to abstain from performing any reduction in order to ensure condition well-founded. In the interest of readability, the search stack parameter is not included in the notation.

During our attempts at formalizing this algorithm, we discovered that one of the lemmata it relies on does not hold. A detailed counterexample for this lemma is presented in appendix A. This counterexample only affected the particular proof and did not contradict the correctness of the algorithm itself. Indeed, at this point, we still believed the algorithm to be correct and thus tried to find a different proof. During these efforts, we discovered another potential issue where the choices in the formula automaton and the choices in the reduction may not line up. If this were to happen, the reduced product could end up with no accepting runs even though the regular product contains some. Unfortunately, we were not able to find a concrete example exhibiting this behavior.

Some time later, Stephen Siegel used the Alloy Analyzer [Jac06] to discover a proper example of this happening [Sie19], shown in figure 5.2.

In this example, the partial order reduction algorithm performs a reduction when invoked as ample $p\, y$. The justification is that since this transition does not close a loop, a reduction that defers operation b should be legitimate. With the property φ being stuttering-invariant, the formula automaton A_φ should be able to accommodate this deferment. Unfortunately, this is not the case. It is true that A_φ allows arbitrary stuttering of the transition α via repetition. However, once it nondeterministically changes from state x to state y, no further stuttering is possible. Thus, if a reduction takes place at the same time that A_φ is transitioning from state x to state y, then operation b gets deferred but can then no longer be executed afterwards.



(a) Automata

(b) Interpretation

(c) Reduction

Figure 5.2: Dynamic Partial Order Reduction Counterexample. Figure a shows the system automaton S, the formula automaton A_φ, and the product automaton $S \times A_\varphi$. Figure b shows the interpretation function for S. Figure c shows the reduction function ample. Note that only ample $p\ y$ allows for any reduction. The blue parts in $S \times A_\varphi$ indicate presence in the regular product, but not in the reduced product. In this example, the regular product contains an accepting run, while the reduced product does not.

This scenario hints at the idea that a stuttering-invariant property φ is not enough and a more local property related to the formula automaton A_φ is needed. Indeed, it is possible to make the automaton itself stuttering-invariant [HK96; Ete99; Sie19]. This means that at every point of a path through the automaton, it is possible to introduce or eliminate stuttering. This avoids the previously described situation where nondeterminism in the formula automaton renders further stuttering impossible. With this stronger property, it is then possible to complete the correctness proof of dynamic on-the-fly partial order reduction [Sie19].

In [Sie19], the counterexample from figure 5.2 is shown to change the result in the Spin model checker when partial order reduction is enabled. It is worth noting that this implementation continued to be used for a long time, despite the issue with the proof being either suspected or known [Sie12; Bru14; BL18]. As mentioned previously, the fact that model checkers are trust multipliers makes this issue particularly critical and emphasizes the need for formal verification of these tools.

5.5 Results

The formalization is available in the Archive of Formal Proofs [Bru18]. It builds on our library for transition systems and automata [Bru17b] and is integrated into the CAVA model checker [Esp+14].

This implementation includes a simple modeling language as well as the static analysis facilities required to realize the partial order reduction algorithm. The result is a formally verified and executable model checker that can perform static partial order reduction.

We evaluate the partial order reduction performance on a distributed mutual exclusion algorithm [NTA96]. We compare the number of states explored during model checking between CAVA and Spin, both with and without partial order reduction enabled. While the absolute number of states generated by Spin is much lower, the reduction factor is comparable between CAVA and Spin, showing the effectiveness of the algorithm. For more details, see appendix A.

Chapter 6

Büchi Complementation

Büchi automata are closed under complementation. That is, for any Büchi automaton, it is possible to find another Büchi automaton whose language is the complement of the language of the first one. This fact was established early on by their inventor Julius Richard Büchi [Büc62]. Since then, Büchi complementation has been a popular research topic [SVW87; Saf88; KV01; FKV06; Sch09]. In fact, its popularity even warranted the publication of several meta papers documenting this research [Var07; Tsa+14].

Much of this interest can be attributed to the difficulty of determining the state complexity of this operation. Complementing a nondeterministic finite automaton with n states results in an automaton with at most $\mathcal{O}(2^n)$ states. The lower bound is established via worst-case examples [SS78] and the upper bound via the textbook complementation algorithm [RS59]. Büchi automata on the other hand, have kept researchers busy for decades, with the upper bound being successively lowered from $2^{2^{\mathcal{O}(n)}}$ [Büc62] to $2^{\mathcal{O}(n^2)}$ [SVW87] to $2^{\mathcal{O}(n \log n)}$ [Saf88] to $(6n)^n$ [KV01] to $\mathcal{O}((0.96n)^n)$ [FKV06] and finally to $\mathcal{O}((0.76n)^n)$ [Sch09], matching the lower bound established by [Mic88].

However, Büchi complementation is not just interesting based on theoretical considerations. One application is model checking, which needs an automaton whose language represents the negation of the checked property. This usually requires the property to be supplied as a formula [VW86], a deterministic automaton [Kur94], or an automaton for the negation of the property [Hol97]. This is done because these representations are either already in the right form or easy to negate or complement, respectively [FKV06]. Having access to Büchi complementation, we can instead use arbitrary automata to represent the checked property. Furthermore,

complementation allows deciding language-wise equivalence of Büchi automata. This can be used to test the correctness of algorithms like LTL formula to automaton translation or automaton simplification.

Complementation algorithms and their correctness proofs are complex. Furthermore, when used as building blocks for verification tools, they act as trust multipliers. Both their complexity and their role as trust multipliers make them a valuable target for formalization. Moreover, the ability to use complementation as part of a decision procedure for language-wise equivalence of Büchi automata makes this an ideal application for a verified reference implementation. Thus, we want to formalize Büchi complementation as well as use it to develop a formally verified equivalence checker.

Related work includes GOAL [Tsa+07], which is an unverified tool that implements various complementation algorithms. There is also Spot [Dur+16], which can complement automata as well as check for language-wise equivalence. To the best of our knowledge, there has only been one other attempt at formalizing Büchi complementation due to Stephan Merz. He formalized preliminary work on weak alternating automata [Mer00] and parts of Büchi complementation. However, this only covers the first part of the complementation procedure and was never finished or published.

6.1 Complementation

Let $A = (\Sigma, Q, I, \delta, F)$ be a nondeterministic Büchi automaton and $w \in \Sigma^\omega$ be a word over its alphabet. We follow the rank-based complementation construction described in [KV01]. It consists of three major components.

Run DAG A directed acyclic graph $G = (V, E)$ whose nodes $V \subseteq Q \times \mathbb{N}$ are pairs of states and natural numbers. Intuitively, each node $(q, k) \in V$ represents A being in state q after having read k characters from w. This way, the run DAG contains all possible paths that A can take while reading w. Figure 6.1 shows an example of such a run DAG.

Odd Ranking A function $f :: V \Rightarrow \mathbb{N}$ assigning a natural number rank to each node in the run DAG. Intuitively, the rank of a node indicates the distance to a node from which no more accepting states are visited [FKV06]. Such an odd ranking certifies the rejection of a word by the automaton.

Complement Automaton The complement automaton $\overline{A}$ is designed to nondeterministically search for an odd ranking, accepting if and only if one exists. This way, the complement automaton accepts exactly those words that the original automaton rejects.

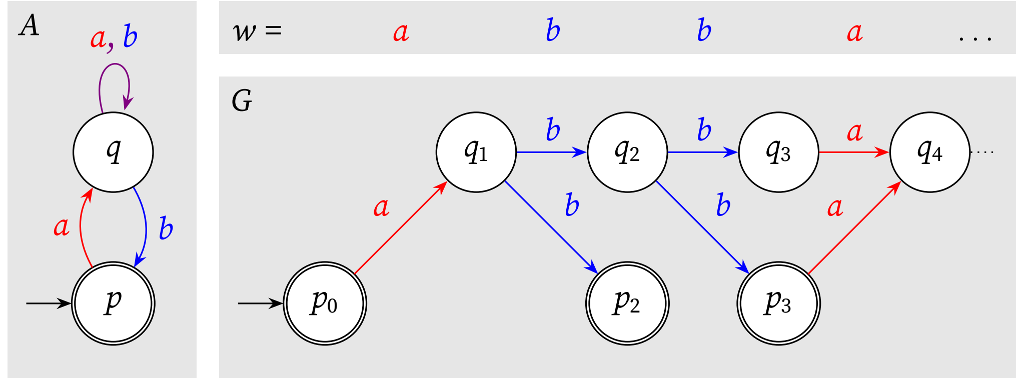


Figure 6.1: Run DAG Example. Shown is an automaton A and its corresponding run DAG G when reading the word $w = abba\ldots \in \Sigma^\omega$. The nodes of the run DAG are pairs (p, k), which are abbreviated to p_k. Note how the run DAG can be partitioned into layers according to node indices with only transitions corresponding to that character in the input word between each layer.

To formalize these concepts, we make heavy use of the library for transition systems and automata introduced in chapter 4. We use it not only to represent Büchi automata, but also for the run DAG. This makes it very easy to relate concepts defined on both of these structures to each other. The formal definition of odd rankings closely resembles the informal one, with only minor differences to account for the implicit use of the definite description operator. Similarly, the formal definition of the complement automaton only differs from the informal one in technical details.

The correctness theorem is then structured as follows.

Theorem 3 (Correctness of Rank-Based Büchi Complementation)

$$w \notin \mathcal{L}\, A \overset{(a)}{\iff} \exists f.\ \text{odd_ranking}\ A\ w\ f \overset{(b)}{\iff} w \in \mathcal{L}\, \overline{A} \tag{6.1}$$

The formal proof of equivalence 6.1a closely follows its informal counterpart. While the proof is substantial, most of the concepts involved could be captured with the tools provided by the distribution and the automata library. One notable exception was the piecewise construction of infinite paths that is described in detail in appendix B. The formal proof of equivalence 6.1b is long and tedious, but mostly technical in nature.

For the implementation, we make extensive use of the refinement framework described in section 2.3. We also use automatic refinement [Lam13a; Lam13b] to derive efficiently executable definitions. Finally, we use the algorithms described in section 4.6 to transform the complement automaton into an explicit representation using natural numbers as the state type.

6.2 Equivalence

Complementation can be used to decide language containment for Büchi automata. To do so, we need an intersection operation and an emptiness check for Büchi automata. We can then decide language containment.

$$\mathcal{L}\,A \subseteq \mathcal{L}\,B \iff \mathcal{L}\,A \cap \overline{\mathcal{L}\,B} = \emptyset \iff \mathcal{L}\,(A \times \overline{B}) = \emptyset \tag{6.2}$$

Checking for containment in both directions leads to a decision procedure for language-wise equivalence of Büchi automata.

The intersection operation was added to the library for transition systems and automata introduced in chapter 4 as part of this project. For the emptiness check, we use a procedure based on Gabow's algorithm for strongly connected components [Gab00]. Compared to emptiness checks based on nested depth-first search, this has the advantage of also working for generalized Büchi automata. This allows us to skip the second step of our intersection operation, which improves performance significantly.

Again, we use the refinement framework for the implementation as well as automatic refinement [Lam13a; Lam13b] to derive efficiently executable definitions. The implementation of the intersection operation was done as part of the automata library and is mostly automatic. The emptiness check based on Gabow's algorithm is available as an AFP entry [Lam14c; Lam14b] originally developed as part of the CAVA model checker. It includes the correctness proof as well as an executable implementation.

6.3 Results

The formalization is available in the Archive of Formal Proofs [Bru17a]. It builds on our library for transition systems and automata [Bru17b].

This includes a command-line tool that provides complementation and equivalence checking of Büchi automata. This tool consists mainly of SML code generated from our theories via Isabelle's code export facilities. The only unverified part is a hand-written parser and printer for the Hanoi Omega-Automata format [Bab+15], also called HOA.

We evaluate the performance of this tool by comparing it against Spot and GOAL. For this, we use randomly generated automata as well as those generated by the LTL translation procedures of Spot and Owl [KMS18]. We also test it on automata generated from known LTL formulae [DAC98]. Our tool is slower than Spot, which is unverified and uses more advanced

algorithms and heuristics. However, it is faster than an implementation of the same algorithm in GOAL, despite the latter being unverified. This is due to more efficient data structures courtesy of the Isabelle Collections Framework [Lam09; LL10]. For more details, see appendix B.

Overall, our tool is sufficiently fast to be useful as a verified reference implementation. For instance, during the aforementioned evaluation, it discovered that for some LTL formulae, Spot and Owl generate non-equivalent automata. The developers of Owl confirmed that this was indeed a bug in the implementation of its translation procedure and promptly fixed it.

Chapter 7

Conclusion

In this dissertation, we set out to use interactive theorem proving to formally verify algorithms associated with model checking. The main advantage of formally verified theories and algorithms is that they are more trustworthy than their pen and paper counterparts. Additionally, the formal proof serves as a machine-checkable certificate of their correctness. Furthermore, the formal theory constitutes a very detailed description of the definitions and proofs involved. Thus, the theory is guaranteed to be properly documented without any missing steps.

As a prerequisite of the algorithms that we wanted to verify, we first formalized a library for transition systems and automata. It constitutes a general and versatile theory that can be reused in and to the benefit of other formalization projects. This is an important aspect, since lack of library support can be a major obstacle for formalization projects. Thus, a growing library of formalizations like this one enables more and more ambitious projects in the future as the foundation that they can build on grows.

Next, we used our library for transition systems and automata to formalize several verification algorithms. Firstly, we formalized a static ample set partial order reduction algorithm and integrated it into the CAVA model checker. Secondly, we formalized a rank-based Büchi complementation algorithm and used it to formalize a Büchi equivalence checker.

The immediate advantage of this is having access to verification tools that are formally verified and thus come with a strong correctness guarantee. This correctness guarantee covers the abstract algorithm as well as its implementation. It is an important ingredient towards building confidence and trust in the verification tools that people use.

Furthermore, these tools can then act as verified reference implementations to check other unverified tools against. This is particularly important in the case of model checkers, where the size of realistic systems often makes it impossible to manually verify the results. Verified reference implementations are especially useful when they concern automata algorithms like determinization or LTL to Büchi translation. These can be used to independently test the building blocks that verification tools are comprised of and thus achieve better modularity and more fine-grained testing.

Finally, Büchi complementation takes on a unique role among automata algorithms. In addition to serving as a reference implementation of complementation algorithms, it can also be used to decide language-wise equivalence of Büchi automata. A formally verified equivalence checker can thus check whether two algorithms produce equivalent automata for a given input. This way, the results of individual automata algorithms can be examined directly rather than treating the model checker as a whole like a black box and comparing only the overall binary result.

All in all, formally verified reference implementations are very useful. They allow transferring some of the confidence and trustworthiness granted by interactive theorem proving to unverified model checking tools.

The importance of formal proofs is often downplayed on the premise that these theorems have already been proven informally. However, formalization efforts have repeatedly proven useful, often leading to the discovery of mistakes in proofs, and sometimes even in theorems. In the case of verified reference implementations, these tools can themselves potentially help uncover issues in the unverified tools that are compared against them.

We believe that our results also serve as a demonstration of the capabilities of interactive theorem proving itself. Not only were we able to formalize nontrivial algorithms, but one of them was integrated into a much larger project, the CAVA model checker. This highlights the software engineering aspect of interactive theorem proving, with modern systems supporting the kind of modularity that makes large endeavors like this possible. This, in turn, allows other projects to reuse these formalization efforts, thus making the system more powerful for everyone.

At this point, the library for transition systems and automata is well-suited to support further formalization projects involving automata. For instance, an algorithm for Büchi determinization would be an obvious continuation of the current line of research. It would add another algorithm to the body of formalized automata theory as well as enable determinization-

based complementation procedures. While the asymptotic complexity of determinization-based complementation is no better than that of the rank-based one, the former promises better performance in practice [Tsa+14].

Another possible avenue would be to formalize the adjustments for dynamic on-the-fly partial order reduction presented in [Sie19]. We estimate that formalizing the notion of stuttering-invariant automata as well as how this property is ensured and affects partial order reduction would be fairly feasible. However, due to the persisting monolithicity of the dynamic partial order reduction algorithm, verifying a model checker employing this algorithm would require a significant amount of work.

The quest for software correctness is a never-ending one, with numerous struggles on all fronts. Particularly within the field of formal verification, there is always more to do. Despite this, we hope that our efforts have had a positive impact both on the people formalizing verification tools and those using the ones we developed.

Part II

Relevant Publications

Appendix A

Partial Order Reduction

This appendix includes a full copy of the following publication.

Julian Brunner and Peter Lammich. "Formal Verification of an Executable LTL Model Checker with Partial Order Reduction". In: *Journal of Automated Reasoning* 60.1 (2018), pp. 3–21. DOI: 10.1007/s10817-017-9418-4

This article is included as part of this publication-based dissertation.

The article above is an extended version of the following publication.

Julian Brunner and Peter Lammich. "Formal Verification of an Executable LTL Model Checker with Partial Order Reduction". In: *NASA Formal Methods - 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings*. Ed. by Sanjai Rayadurgam and Oksana Tkachuk. Vol. 9690. Lecture Notes in Computer Science. Springer, 2016, pp. 307–321. DOI: 10.1007/978-3-319-40648-0_23

Copyright

The journal article in this appendix was published by Springer Nature. Per author retained rights, it can be reused in dissertations without obtaining permission from the publisher. The accepted version of the manuscript is thus reproduced here in accordance with the requirements of the publisher.

Summary

This article describes the formal verification of a partial order reduction algorithm using the proof assistant Isabelle. We follow the ample set approach to partial order reduction [Pel96]. This method introduces a set of reduction conditions that form the basis of the abstract correctness proof. The actual reduction algorithm that ensures that these reduction conditions hold is based on a static analysis approach [Kur+98].

We start by formalizing several prerequisites on finite, infinite, and mixed sequences, the last of which we extend with limits and index sets. These are then used to formalize the parts of trace theory [Maz86] required for partial order reduction. We also formalize a library for systems with named operations and connect it to our formalization of trace theory. With these prerequisites, we then formalize the abstract correctness proof of partial order reduction that is given in [Pel96].

We formalize a fragment of PROMELA together with the static analysis required for partial order reduction. We then define the actual reduction algorithm based on this, proving that it ensures the reduction conditions that the abstract correctness proof needs. After this, we use refinement to derive an executable implementation from this algorithm.

This algorithm is then integrated into the CAVA model checker, resulting in a formally verified and executable on-the-fly LTL model checker with partial order reduction. As all aspects are integrated into one formalization, the verification covers everything from the abstract correctness proofs down to the generated SML code. We evaluate the effectiveness of the model checker on a distributed mutual exclusion algorithm.

Finally, we present a counterexample for one of the lemmata involved in the proof of dynamic on-the-fly partial order reduction.

Contributions

I am the first author of this article. My contributions are the formalization of abstract partial order reduction as well as prerequisites with respect to sequences, trace theory, and automata. I also contributed a framework for the static analysis part of the reduction algorithm. Finally, I contributed the counterexample for the dynamic on-the-fly partial order reduction proof.

Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Formal Verification of an Executable LTL Model Checker with Partial Order Reduction

Julian Brunner · Peter Lammich

the date of receipt and acceptance should be inserted later

Abstract We present a formally verified and executable on-the-fly LTL model checker that uses ample set partial order reduction. The verification is done using the proof assistant Isabelle/HOL and covers everything from the abstract correctness proof down to the generated SML code.

Building on Doron Peled's paper "Combining Partial Order Reductions with On-the-Fly Model-Checking", we formally prove abstract correctness of ample set partial order reduction. This theorem is independent of the actual reduction algorithm. We then verify a reduction algorithm for a simple but expressive fragment of Promela. We use static partial order reduction, which allows separating the partial order reduction and the model checking algorithms regarding both the correctness proof and the implementation. Thus, the Cava model checker that we verified in previous work can be used as a back end with only minimal changes. Finally, we generate executable SML code using a stepwise refinement approach. We test our model checker on some examples, observing the effectiveness of the partial order reduction algorithm.

1 Introduction

Partial order reduction [28] is an important optimization for model checkers, enabling them to deal better with models involving concurrency. It allows the model checker to consider only a subset of all possible interleavings of concurrently executing operations by identifying equivalences between them. Unfortunately, partial order reduction is notoriously complex and can easily affect the correctness of the model checker. For instance, [28] describes a partial order reduction algorithm and claims that it can simply be used with on-the-fly nested depth-first search. It was found out later that this compromises correctness due to the reduction possibly differing between the inner and the outer search [9]. Moreover, while formalizing the algorithm in [28], we discovered that its correctness proof uses an invalid lemma.

Implementation correctness is usually assessed via testing in the context of model checking algorithms. However, testing is necessarily incomplete and may lead to incorrect

Research supported by DFG grant Cava (Computer Aided Verification of Automata, ES 139/5-1, NI 491/12-1, SM 73/2-1) and Cava2 (Verified Model Checkers, KR 4890/1-1, LA 3292/1-1)

Technische Universität München

implementations due to missed corner cases. Furthermore, when using models of realistic size, determining the correct outcome for a given test input requires the use of a model checker.

Thus, although in widespread use, neither the correctness of partial order reduction algorithms, nor the correctness of their implementations can be taken for granted. This is especially problematic since the trust in the correctness of a single model checker is used to justify the confidence in the correctness of the many models that it checks. In order to meet the very strict correctness requirements of model checking algorithms, we implement and formally verify a partial order reduction algorithm.

In previous work [6], we have presented the CAVA model checker, a fully verified and executable LTL model checker à la SPIN. The verification was done with the proof assistant Isabelle/HOL [27] and covers everything from the correctness of the algorithms down to the implementation. Due to its LCF-like architecture, Isabelle/HOL is more trustworthy than a large unverified implementation like SPIN. This paper now adds the following contributions:

1. Formalization of a fragment of the modeling language PROMELA
2. Formalization of the static analysis required for partial order reduction
3. Formal abstract correctness proof for ample set partial order reduction
4. Verified implementation and integration into the CAVA model checker
5. Development of reusable libraries for automata and trace theory

This results in what we believe to be the first formally verified and executable implementation of partial order reduction, addressing both of the issues mentioned earlier. The verification is carried out completely in Isabelle/HOL, such that the correctness of the model checker only depends on the correctness of Isabelle/HOL. This integration avoids logical gaps that may arise when manually composing the results of different verification tools. Most importantly, we now have a formally verified reference implementation that can deal with many models that would be infeasible without partial order reduction. This improves its usefulness for testing other model checkers. To the best of our knowledge, there has been only one other attempt at formalizing partial order reduction [5]. However, it does not cover the reduction algorithm and is restricted to a specific fairness assumption.

The rest of the paper is organized as follows. In section 2, we cover theoretical aspects of partial order reduction and elaborate on our choice of algorithm. In section 3, we report on our Isabelle/HOL formalization. In section 4, we test the reduction effectiveness of our implementation. Finally, in section 6, we give conclusions and future research directions.

This paper is an extended version of [4]. It includes more details about the formalization of the abstract correctness proof of partial order reduction in section 3.6. There is also the new section 3.9 describing the architecture of the CAVA model checker. Finally, we added section 5. It describes a counterexample for the invalid lemma used in the correctness proof of dynamic partial order reduction with on-the-fly model checking.

2 Theory

Figure 1 illustrates the basics of partial order reduction. In regular model checking, the system automaton 'S' is derived from the system and used as input for the model checker together with the formula 'φ'. The model checker then determines if the system automaton satisfies the property expressed by the formula ($\mathcal{L}\ S \subseteq \mathcal{L}\ \varphi$). When using partial order reduction, a reduction algorithm obtains a reduced system automaton 'R' from the system instead, which fulfills certain *reduction conditions*. These conditions imply stuttering equivalence between the language of the system automaton and that of the reduced system automaton

($\mathcal{L}\ S \approx \mathcal{L}\ R$). Since properties expressed by next-free LTL formulae are stuttering-invariant [29], using the reduced system automaton instead of the system automaton when model checking yields the same result ($\mathcal{L}\ S \subseteq \mathcal{L}\ \varphi \iff \mathcal{L}\ R \subseteq \mathcal{L}\ \varphi$).

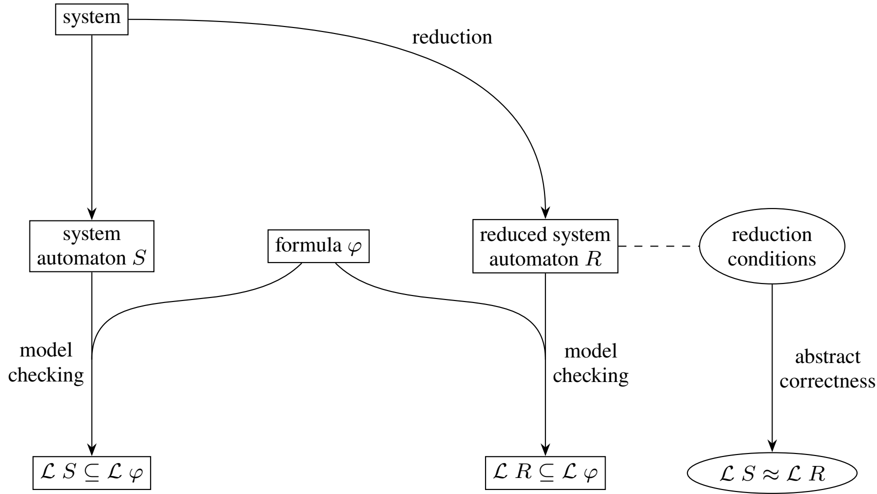


Fig. 1: Partial Order Reduction Overview. A reduction algorithm obtains the reduced system automaton 'R', which is then used as an input of the model checker instead of the system automaton 'S'. The reduction algorithm guarantees that the reduced system automaton fulfills certain reduction conditions, from which one can prove stuttering equivalence between the two languages. This implies that the result of the model checker is not affected by the reduction.

This is a very abstract description of partial order reduction. In actual implementations, the reduced system automaton may be represented implicitly, and the reduction algorithm may be merged with the model checking algorithm. However, this view allows us to identify the three major tasks involved in developing a verified implementation of partial order reduction:

1. Reduction algorithm correctness: The automaton produced by the reduction algorithm fulfills the reduction conditions.
2. Abstract correctness: If an automaton fulfills the reduction conditions, its language is stuttering equivalent to that of the system automaton.
3. Implementation and verification of the reduction algorithm.

Unlike our formalization, [5] only covers the second task. This means there is no input language, no static analysis, no reduction algorithm, no implementation, and no executable model checker. Furthermore, it only covers the case where a certain fairness assumption is met, which simplifies the abstract correctness proof. In absence of other formalizations, we believe that our work is a significant contribution over the existing body of research.

2.1 Reduction Conditions

Both the reduction algorithm and the abstract correctness are built around the reduction conditions, making them the main object of interest when dealing with partial order reduction. We chose to implement an algorithm based on the ample set method and chose the reduction

conditions accordingly. Let '$\text{en } q$' be the set of enabled actions at state 'q' of the system automaton (*enabled set*). Let '$\text{ren } q$' be the set of enabled actions at state 'q' of the reduced system automaton (*ample set*). Let '$\text{ex } a\ q$' be the successor of state 'q' after executing action 'a' ('ex' is called *execution function*). This way, the pair '(en, ex)' represents the system automaton, while '(ren, ex)' represents the reduced system automaton. The set of finite paths executable at state 'q' of the system automaton '$\text{paths } q$' is defined in terms of 'en' and 'ex'. For a more detailed description of the system definitions, see section 3.4. With these prerequisites, we define the following reduction conditions:

subset	$\forall q.\ \text{ren } q \subseteq \text{en } q$
nonempty	$\forall q.\ \text{ren } q \subset \text{en } q \implies \text{ren } q \neq \{\}$
independent	$\exists$ independence relation $I.\ \forall q\ w.\ \text{ren } q \subset \text{en } q \implies$ $w \in \text{paths } q \implies \text{ren } q \cap \text{set } w = \{\} \implies I\ (\text{ren } q)\ (\text{set } w)$
wellfounded	$\exists$ well-founded relation $R.\ \forall q\ a.\ \text{ren } q \subset \text{en } q \implies$ $a \in \text{ren } q \implies R\ (\text{ex } a\ q)\ q$
invisible	$\forall q.\ \text{ren } q \subset \text{en } q \implies \text{ren } q \subseteq \text{invisible}$

Condition **subset** states that the reduced system automaton is a subautomaton of the system automaton and is usually not stated explicitly in the literature. Condition **nonempty** states that the reduction algorithm must not omit all of the actions at any state. Condition **independent** requires that all the actions that are executed after reaching some state but before an action from the ample set at this state are *independent* of all the actions in this ample set. Condition **wellfounded** requires that every cycle in the system automaton contains at least one state where no reduction is performed. Condition **invisible** states that when a proper reduction takes place, the ample set cannot contain any actions that are *visible* to the formula. Conditions **nonempty**, **independent**, and **wellfounded** correspond to conditions C0, C1, and C2 in [5, pages 268, 269]. Condition **invisible** corresponds to condition $C3'$ in [28, page 50]. Note that even though the reduction conditions are similar, our formalization is not based on [5].

2.2 Reduction Algorithm

These conditions are very abstract, so there are still many choices to be made with respect to the actual reduction algorithm. We originally planned to verify dynamic partial order reduction with on-the-fly model checking [28], but soon encountered difficulties. Dynamic partial order reduction detects cycles during the emptiness check in order to ensure condition **wellfounded**. This tight integration with the emptiness check has led to bugs in the past [9]. When used with on-the-fly model checking, this integration also extends to the product construction, effectively turning the whole model checker into one monolithic algorithm. It also introduces a mismatch since an algorithm that conceptually works on a system automaton is now used with a product automaton, requiring complicated reasoning. And indeed, during our effort of formalizing the proof given in [28], we discovered a counterexample for one of the lemmata used in this proof. This counterexample is based on the fact that, when exploring the product automaton, different instances of the system automaton appearing in the product automaton may be reduced differently. A detailed description can be found in section 5. Note that this, while refuting the lemma, does not necessarily invalidate the correctness theorem, only this particular proof thereof. However, despite investing a significant amount of time, we were unable to find an alternative proof as it seems that the reasoning required is more complex than anticipated in the original paper.

We chose to implement a static partial order reduction [10] algorithm instead, which avoids these problems of the dynamic approach. It ensures condition **wellfounded** by performing

some static analysis initially, identifying a set of *sticky* edges which break every cycle in the control flow graph. Static partial order reduction is much more modular, making it possible to verify the reduction algorithm independently of the product construction and the emptiness check. This way, we were able to simply add the reduction algorithm as a preprocessing step to the existing CAVA model checker, enabling reuse of existing optimizations.

The reduction algorithm itself is similar to the one used in SPIN [8]. The basic idea is to take the set of enabled actions of each process at some state as a candidate for an ample set. For each candidate, an over-approximation of the reduction conditions is tested. If no candidate satisfies the conditions, the state is fully expanded, that is, no reduction is performed.

For instance, our approximation checks that, in order to be used as an ample set, the actions of a process must be independent of all actions of other processes. Moreover, it is checked that no additional action of this process can be enabled as a consequence of executing actions of other processes. Thus, only independent actions of other processes can be executed before an action of the ample set, which implies condition **independent**.

3 Formalization

Our formalization contains all three of the tasks outlined in section 2. We integrated our implementation into the CAVA model checker, which was published previously [6, 7]. Since then, various features have been added to this model checker. For instance, it now supports using PROMELA as an input language [25]. Furthermore, the library for automata has been updated [20, 14] and a new framework for depth-first search algorithms has been formalized [18]. Also, an alternative algorithm for deciding language emptiness of Büchi automata based on Gabow's strongly-connected components algorithm has been implemented [16]. In order to make all of these changes possible, the architecture of the CAVA model checker was improved to be more modular and extensible (see section 3.9). However, the focus of this paper is on the implementation and verification of the partial order reduction algorithm.

In this section, we give some technical background regarding the tools that were used as well as a high-level overview of the formalization. We also describe certain noteworthy aspects of the formalization in isolated detail. The full formalization is available at `https://cava.in.tum.de/CAVA_POR`.

3.1 Isabelle/HOL

Isabelle/HOL [27, 26] is a proof assistant based on Higher-Order Logic (HOL), which can be thought of as a combination of functional programming and logic. Formalizations done in Isabelle/HOL are trustworthy for two reasons. Firstly, Isabelle's LCF architecture guarantees that all proofs are checked using a very small logical core which is rarely modified but tested extensively over time. This reduces the trusted code base to a minimum. Secondly, bugs in the core rarely lead to accidentally proving false propositions. Bugs that have large effects are easily caught, while the limited applicability of bugs with small effects is unlikely to coincide with a logical mistake in the large-scale structure of the proof.

Isabelle/HOL notation resembles standard mathematical notation with just a few differences. For instance, as in functional programming, functions are usually curried in HOL. This means that instead of '$f :: A \times B \to C$' with application syntax '$f(x, y)$', we have '$f :: A \to B \to C$' with application syntax '$f\ x\ y$'.

3.2 Refinement Framework

We want our model checker and the partial order reduction algorithm contained therein to be executable. When developing formally verified algorithms, there is a trade-off between the efficiency of the algorithm and the efficiency of the proof: For complex algorithms, a direct proof of an efficient implementation tends to get unmanageable, as implementation details obfuscate the main ideas of the proof. A standard approach to this problem is stepwise refinement [1], which modularizes the correctness proof: One starts with an abstract version of the algorithm and then refines it in correctness-preserving steps to the concrete, efficient version. A refinement step may reduce the nondeterminism of a program, replace abstract mathematical specifications by concrete algorithms, and replace abstract datatypes by their implementations. For example, selection of an arbitrary element from a set may be refined to getting the head of a list. This approach separates the correctness proof of the algorithm, which focuses on the main algorithmic ideas, from the correctness proof of the implementation, where the proof of each refinement step focuses on a specific implementation detail, not caring about the overall correctness property.

In Isabelle/HOL, stepwise refinement is supported by the Refinement Framework [19, 12, 13, 15] and the Isabelle Collection Framework [17, 11]. The former framework implements a refinement calculus [1] based on a nondeterminism monad [30], and the latter provides a library of verified efficient data structures. Both frameworks come with tool support to simplify their usage for algorithm development and to automate canonical tasks such as verification condition generation.

3.3 Basics

The most basic concept needed for nearly all parts of the formalization is that of *sequences*. With HOL being very similar to functional programming languages like SML or Haskell, the standard library already includes extensive support for *finite sequences* via the type 'α list $=$ Nil | Cons α (α list)'. For *infinite sequences*, the type 'α word' is used, which is simply a type synonym for '$\mathbb{N} \to \alpha$'.

We also use the library Coinductive [21] which formalizes lazy lists using codatatypes [2]. It provides the type 'α llist', which models both finite and infinite sequences. This is useful for selecting subsequences of infinite lists that can be either finite or infinite. Reasoning about selections and indices of lazy lists required us to significantly extend the library Coinductive.

Another important component needed for partial order reduction is stuttering equivalence and the proof that next-free LTL formulae can only express stuttering-invariant properties. The library Stuttering Equivalence [23] is used for both.

3.4 Systems

Model checkers usually represent systems using the type '$(\text{state} \times \text{state})\ \text{set}$'. Reasoning about partial order reduction requires transitions to be labeled with actions, suggesting the type '$(\text{state} \times \text{action} \times \text{state})\ \text{set}$'. However, this type allows multiple successor states to be reached given a state and an action, making the type a bad fit for the deterministic action model of partial order reduction. This leads to unnecessary wellformedness conditions, inaccessible successor states, and overspecified path predicates. We thus chose the following

representation of the system automaton which was already referred to in section 2.1:

$$\text{en} :: \text{state} \rightarrow \text{action set} \tag{1a}$$
$$\text{ex} :: \text{action} \rightarrow \text{state} \rightarrow \text{state} \tag{1b}$$
$$\text{init} :: \text{state set} \tag{1c}$$

Here, 'en' is the set of enabled actions at a state (*enabled set*), 'ex' is the function that, given an action, maps each state to its successor state (*execution function*), and 'init' is the *set of initial states*.

This representation allows paths to be introduced in a straightforward way via the inductively defined set '$\text{paths} :: \text{state} \rightarrow \text{action list set}$':

$$[] \in \text{paths } p \tag{2a}$$
$$a \in \text{en } p \implies w \in \text{paths } (\text{ex } a\ p) \implies a \# w \in \text{paths } p \tag{2b}$$

Inductive definitions specify the smallest sets that satisfy the given rules. Equivalently, they specify the sets containing those elements whose membership can be derived using the given rules. These rules can be declared as safe introduction rules, so that whenever Isabelle/HOL encounters proof obligations of the form '$[] \in \text{paths } p$' or '$a \# w \in \text{paths } p$', it can automatically split them into simpler goals or discharge them completely.

We prove an additional rule for the append operator on lists:

$$u \in \text{paths } p \implies v \in \text{paths } (\text{fold ex } u\ p) \implies u @ v \in \text{paths } p \tag{3}$$

Note how 'fold' lifts the execution function '$\text{ex} :: \text{action} \rightarrow \text{state} \rightarrow \text{state}$' from single actions to sequences of actions '$\text{fold ex} :: \text{action list} \rightarrow \text{state} \rightarrow \text{state}$'. Also note how this rule generalizes rule 2b.

Together, rules 2a, 2b, and 3 form a set of introduction rules that break down most goals automatically. For instance, the goal '$u @ a \# v \in \text{paths } p$' gets transformed into three subgoals:

$$u \in \text{paths } p \tag{4a}$$
$$a \in \text{en } (\text{fold ex } u\ p) \tag{4b}$$
$$v \in \text{paths } (\text{ex } a\ (\text{fold ex } u\ p)) \tag{4c}$$

Compared to the formalization using the type '$(\text{state} \times \text{action} \times \text{state})\ \text{set}$', this automates proofs significantly. In some cases, proofs comprised of 50 to 100 lines become one-liners. We have proven many more rules about this system formalization, making it a useful addition to the CAVA automata library.

3.5 Trace Theory

In order to formalize partial order reduction, we need the concept of *independent* actions, which can be executed in any order without changing the result or enabling or disabling each other. Trace theory [22] lifts this notion of commutable items to that of *equivalent* ($\equiv_I$) sequences, which is needed in the abstract correctness proof.

Finite sequences are equivalent if they differ by a finite number of commutations of independent actions. Using equivalence on finite sequences, it is possible to define equivalence on infinite sequences via a series of definitions [28, page 41]. Unfortunately, these definitions

are not easily generalized to lazy lists, so we decided to work with separate types and definitions for finite and infinite sequences.

Formalizing the necessary parts of trace theory took significant effort due to the large number of theorems. There are also some theorems that look simple but are difficult to prove, for instance:

$$w_1 \equiv_I w_2 \iff u \mathbin{@} w_1 \mathbin{@} v \equiv_I u \mathbin{@} w_2 \mathbin{@} v \tag{5}$$

The left to right direction can be proven via rule induction on the transitive structure of '$\equiv_I$'. Doing the same for the right to left direction results in an unprovable induction step. It was necessary to prove the following lemmata:

$$w_1 \equiv_I w_2 \implies \text{remove1 } c\ w_1 \equiv_I \text{remove1 } c\ w_2 \tag{6a}$$

$$u \mathbin{@} w_1 \equiv_I u \mathbin{@} w_2 \implies w_1 \equiv_I w_2 \tag{6b}$$

$$w_1 \equiv_I w_2 \implies \text{rev } w_1 \equiv_I \text{rev } w_2 \tag{6c}$$

Here, '$\text{remove1 } c\ w$' removes the first occurrence of 'c' from the sequence 'w', and '$\text{rev } w$' reverses the sequence 'w'. Lemma 6a uses 'remove1' to avoid the fact that rule induction does not work with modified assumptions. We use lemma 6a to prove lemma 6b via reverse induction on the sequence 'u'. Lemma 6c is proven via rule induction and with lemma 6b, it completes the proof of theorem 5.

We also had to define some concepts specific to partial order reduction. For instance, the predicate specifying that the first occurrence of a symbol in a sequence is independent of all symbols before it. In the end, the formalization of the relevant aspects of trace theory required about as much proof text as the formalization of the abstract correctness proof itself.

3.6 Abstract Correctness

In this section, we discuss the part of the formalization dealing with the abstract correctness proof of ample set partial order reduction. Assume that 'S' is a system automaton and 'R' is a reduced system automaton such that the reduction conditions introduced in section 2.1 hold. Then, the abstract correctness theorem states that the languages of 'S' and 'R' are stuttering equivalent:

$$\mathcal{L}\ S \approx \mathcal{L}\ R \tag{7}$$

The proof of this theorem required about 1000 lines of formal proof text including dozens of lemmata. Its structure is similar to that of the informal proof [28].

However, we present the formalization of a lemma [28, Theorem 3.11] in detail and highlight the differences between the formal and the informal proof. Informally, the lemma states that, given an infinite sequence in the system automaton, it is possible to find a corresponding sequence in the reduced system automaton. We would like to convey an idea of what the formal proof looks like without going into every detail of it.

To do so, we need some definitions and some notation. We use '$\frown$' to denote concatenation of a finite sequence and an infinite sequence. The constant 'Ind' lifts the independence relation 'I' to sets. The operators '$=_F$' and '$=_I$' denote equivalence between finite and infinite sequences, respectively.

First, we construct an arbitrarily long but finite sequence in the reduced system automaton by transcribing longer and longer prefixes of the infinite sequence 'v' in the system automaton. In order to do so, we inductively define a predicate that describes a valid state during this construction process where a prefix of the sequence in the system automaton has already

been processed. We use the command '**inductive**' to define the constant 'reduced_run' as the least predicate that satisfies the rules init, absorb, and extend:

inductive reduced_run :: "state ⇒ action list ⇒ action word ⇒ action list ⇒ action list ⇒ action list ⇒ action list ⇒ action word ⇒ bool" **where**
init: "reduced_run q [] v [] [] [] [] v" |
absorb: "reduced_run $q\ v_1\ ([a] \frown v_2)\ l\ w\ w_1\ w_2\ u \implies a \in \text{set}\ l \implies$ reduced_run $q\ (v_1 @ [a])\ v_2\ (\text{remove1}\ a\ l)\ w\ w_1\ w_2\ u$" |
extend: "reduced_run $q\ v_1\ ([a] \frown v_2)\ l\ w\ w_1\ w_2\ u \implies a \notin \text{set}\ l \implies$
$b @ [a] \in R.\text{paths}\ (\text{fold ex}\ w\ q) \implies \text{Ind}\ \{a\}\ (\text{set}\ b) \implies \text{set}\ b \subseteq \text{invisible} \implies$
$b =_F b_1 @ b_2 \implies [a] \frown b_1 \frown u' =_I u \implies \text{Ind}\ (\text{set}\ b_2)\ (\text{range}\ u') \implies$
reduced_run $q\ (v_1 @ [a])\ v_2\ (l @ b_1)\ (w @ b @ [a])\ (w_1 @ b_1 @ [a])\ (w_2 @ b_2)\ u'$"

The predicate 'reduced_run' uses the following parameters:

q	initial state for 'v'
v_1	prefix of 'v' that has been processed so far
v_2	suffix of 'v' that has not yet been processed
l	actions that have been appended to 'w_1' but did not yet appear in 'v_1'
w	sequence in the reduced system automaton constructed so far
w_1	possibly visible part of 'w'
w_2	invisible part of 'w'
u	possible continuation of 'w'

This predicate specifies that the state of the construction where nothing has been processed yet is valid (init). It also specifies how one can extend a valid construction state by adding a step in the system automaton and a sequence of corresponding steps in the reduced system automaton (absorb and extend).

Next, we present some theorems that can be proven about this predicate using Isabelle syntax, which should be fairly intuitive. After the theorem name, assumptions are stated using the '**assumes**' directive and the conclusion is stated using the '**shows**' directive. The notation '**obtains** x **where** P' denotes that the theorem proves an existential statement of the form '$\exists\, x.\ P$'. We first prove that certain invariants hold at each point of the construction:

lemma reduced_run_invariants:
assumes "reduced_run $q\ v_1\ v_2\ l\ w\ w_1\ w_2\ u$"
shows "$w \in R.\text{paths}\ q$" "$v_2 =_I l \frown u$" "$v_1 @ l =_F w_1$" "$w =_F w_1 @ w_2$"
"filter visible w_1 = filter visible w" "set $w_2 \subseteq$ invisible"
"Ind (set w_2) (range u)" "length $v_1 \leq$ length w_1" "length $v_1 \leq$ length w"

We also prove that the construction can always be extended:

lemma reduced_run_step :
assumes "$q \in$ reachable" "$v_1 \frown [a] \frown v_2 \in$ runs q"
assumes "reduced_run $q\ v_1\ ([a] \frown v_2)\ l\ w\ w_1\ w_2\ u$"
obtains $l'\ w'\ w_1'\ w_2'\ u'$
where "reduced_run $q\ (v_1 @ [a])\ v_2\ l'\ (w @ w')\ (w_1 @ w_1')\ (w_2 @ w_2')\ u'$"

Proving reduced_run_invariants and reduced_run_step required a lot of effort as the informal proof only provided a rough sketch of the arguments underlying these proofs.

With these lemmata proven, we can now show our lemma [28, Theorem 3.11]:

lemma reduction_word :
  assumes "$q \in$ reachable" "$v \in$ runs q"
  obtains $u\ w$
  where "$w \in R$.runs q" "$v =_I u$" "$u \preceq_I w$"
    "lfilter visible (inf_llist u) = lfilter visible (inf_llist w)"
proof –
  def $P \equiv$ "$\lambda\, k\ w\ w_1.\ \exists\, l\ w_2\ u.$ reduced_run q (prefix $k\ v$) (suffix $k\ v$) $l\ w\ w_1\ w_2\ u$"
  obtain $w\ w_1$ **where** "$\forall\, k.\ P\ k\ (w\ k)\ (w_1\ k)$" "chain w" "chain w_1" ...
  proof (rule chain_construct_2' [of P])
    show "$P\ 0$ [] []" ...
  next
    fix $k\ w\ w_1$
    assume "$P\ k\ w\ w_1$"
    show "$\exists\, w'\ w_1'.\ P\ (Suc\ k)\ w'\ w_1' \wedge w \leq w' \wedge w_1 \leq w_1'$" ...
    show "$k \leq$ length w" "$k \leq$ length w_1" ...
  qed rule
  show ?thesis
  proof
    show "limit $w \in R$.runs q" ...
    show "$v =_I$ limit w_1" ...
    show "limit $w_1 \preceq_I$ limit w" ...
    show "lfilter visible (inf_llist (limit w_1)) = lfilter visible (inf_llist (limit w))"
      ...
  qed
qed

Here, 'lfilter' is the filter function on lazy lists and 'inf_llist' converts an infinite sequence to a lazy list. Proofs are enclosed in '**proof** ... **qed**' blocks, with '**next**' separating subgoals. Inside these blocks, arbitrary propositions can be proven. Existential statements use the form '**obtain** ... **where**'. Local definitions can also be made via '**def**'. The directives '**fix**', '**assume**', and '**show**' are used to work with universally quantified variables, assumptions, and goals, respectively. Once all goals have been discharged via '**show**', the proof block can be closed via '**qed**'. Note that we used '...' to signify an omitted proof.

In the proof, we have to show that there exists an infinite sequence 'w' with the required properties in the reduced system automaton. While this step is almost completely skipped in the informal proof, the formal one forces us to consider it rigorously. Lemma reduced_run_step guarantees that for any number of steps that were already taken, another step can be taken, extending the sequence in the process. Intuitively, such a theorem can be applied "infinitely often" to obtain an infinite sequence, but this is not logically sound. Performing a step like

this in a formal proof requires precise reasoning and in our case the use of Hilbert's epsilon operator in lemma chain_construct_2'. This lemma turns the ability to perform an arbitrary number of steps into an infinite chain of finite sequences where each sequence is a prefix of the one following it. This property is stated via the constant 'chain'. The constant 'limit' is then used to derive the uniquely determined infinite sequence from this chain.

We believe that these difficulties do not point to a shortcoming of formal logic or the particular system we are using. Instead, we think that situations like this one point to areas where it became customary to use sloppy reasoning in informal proofs, possibly leading to mistakes or overlooked side conditions. For instance, it is often not made clear in which way variables depend on each other or what guarantees that an infinite sequence can actually be constructed from an infinite set of finite sequences. Formal proofs point out required side conditions like the fact that the infinite concatenation of these finite sequences needs to be infinite. It also brought attention to the fact that many concepts need to be defined on both finite and infinite sequences and that they need to correspond to each other in a specific way.

The formal proof constitutes both a certificate of the theorem's correctness as well as a detailed documentation of the reasoning used to prove it. As mentioned in sections 3.3, 3.4, and 3.5, a large amount of foundational work was required in order to formally prove the abstract correctness theorem.

3.7 The SM Language

In order to implement an executable reduction algorithm, we require a concrete modeling language. We use a simple fragment of Promela that is expressive enough to model interesting examples. We call this fragment the *SM language* (simple modeling language).

A program in this language consists of a set of processes, each of which is described using a guarded command language. Each process has a set of local variables and communication between processes is modeled via global variables. A configuration of the system consists of a valuation of the global variables and a list of process configurations, where a process configuration consists of a command and a valuation of the local variables.

Most features of Promela are either contained in the SM language or can be expressed directly using global variables. The behavior of channels, small integers, arrays, dynamic processes, and process priorities has to be emulated via more complex constructions using global variables.

We specify a structural operational semantics that establishes a control flow graph where the nodes are commands and the edges are labeled with *local actions*. A local action can be a guarded assignment, a test, or the skip action. Each local action is assigned an enabledness check and an effect function on the local and global variables.

The system semantics describes a step relation between configurations by nondeterministically picking a process from a configuration, following an edge in the control flow graph from the process' command that is labeled with an enabled local action, and applying the effect of the local action to the local and global state. To ensure that all runs of the system are infinite, we apply a stuttering extension, that is, if there is no process with an enabled action, the system may take a step that does not change the configuration.

Since we want to use the SM language in an LTL model checker, we need to define *atomic propositions* and their connection to the system states. In our case, atomic propositions are simply expressions in the SM language that contain only global variables. Then, we define the *interpretation function* to map each state to the set of expressions that evaluate to a non-zero value in this state. Finally, we define the language of a program as the set of infinite sequences

of sets of atomic propositions that correspond to infinite runs of the program:

$$\mathcal{L} :: \text{program} \rightarrow \text{exp set word set} \tag{8}$$

We define a *global action* to consist of a process ID and a control flow graph edge. The process ID is the position of the associated process in the list of all processes. A global action is enabled if the associated process exists, the control flow graph edge is consistent with the current command of the associated process, and the corresponding local action is enabled. Execution of a global action transforms the state of the associated process and the global variables according to the corresponding local action.

3.8 Reduction Algorithm

We make some approximations similar to those made in SPIN in order to define an efficiently executable function which selects an ample set for a configuration, thereby implementing the reduction algorithm. We call an action *statically enabled* if it occurs on a control flow graph edge consistent with the current command of its process in the configuration. This overapproximates the set of enabled actions by ignoring the enabledness conditions.

Similar to SPIN, candidates for ample sets are the sets of enabled actions of each process. We make a crude approximation and allow a nonempty set of enabled actions of a process as an ample set, if (1) there is no statically enabled action of the process that reads or writes global variables, and (2) none of the enabled actions corresponds to a sticky edge in the control flow graph. Here, (1) is a way of guaranteeing condition **independent** (see section 2.1), and (2) is the condition imposed by static partial order reduction (see section 2.2).

We implemented and verified an algorithm based on depth-first search which computes the set of sticky edges before the model checking phase. This algorithm starts with the set of edges labeled with actions containing global variables and extends it to a feedback arc set on the control flow graphs of the processes. For this task, we used the DFS Framework [18], which simplifies the implementation and verification of efficient DFS-based algorithms.

We define the reduced system automaton based on this ample function and prove that all of the reduction conditions from section 2.1 are fulfilled. This allows us to invoke the abstract correctness theorem to obtain stuttering equivalence between the language of the system automaton and that of the reduced system automaton. Together with the assumption that the formula is next-free, this implies that using the reduced system automaton for model checking instead of the system automaton does not change the result.

3.9 Architecture of the CAVA Model Checker

Since the previous publication on the CAVA model checker [6], many improvements have been made. Following an overhaul of the automata library [20, 14], the architecture of the model checker has been generalized to a point where it can be considered a generic framework for assembling formally verified LTL model checkers. It is this architecture that we want to describe in this section. Figure 2 shows the data flow of the CAVA model checker. We use *generalized Büchi graphs* to represent the product automaton, which are generalized Büchi automata over the unit alphabet. As the alphabet is not needed to check for emptiness, this avoids forcing implementations to include alphabet information in their product automaton implementation.

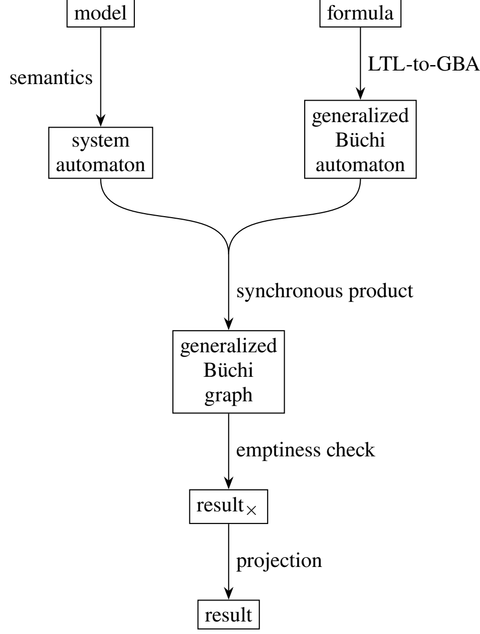


Fig. 2: Architecture of the CAVA Model Checker. The boxes show the data, and the arrows show the operations of the model checker. The input is a model and a formula. The semantics of the modeling language interprets the model as a system automaton, and the LTL-to-GBA conversion creates a generalized Büchi automaton from the formula. Then, the synchronous product of the two is formed, and checked for emptiness. The result is either emptiness or an accepting run of the product automaton. In the latter case, the run is projected back to the system automaton.

In order to get a flexible framework, we specify the data and components on an abstract level. Automata are specified as relations over nodes and a function assigning sets of atomic propositions to nodes. The specification is parameterized over the types of nodes and atomic propositions. We do not specify an abstract model, but start with the system automaton.

The abstract components are nondeterministic specifications of the expected behavior. For example, we specify the emptiness check 'echk_abs' to return 'UNSAT' together with some accepting run '$r_\times$' of the generalized Büchi graph 'G' if there is one, and 'SAT' otherwise:

$$\begin{aligned} &\textbf{case}\ \ \text{echk_abs}\ G\ \ \textbf{of} \\ &\quad \text{SAT} \Rightarrow G.\text{runs} = \{\}\ | \\ &\quad \text{UNSAT}\ r_\times \Rightarrow r_\times \in G.\text{runs} \end{aligned} \tag{9}$$

We assemble the abstract components to form the abstract model checker 'cava_abs' and show its correctness:

$$\begin{aligned} &\textbf{case}\ \ \text{cava_abs}\ S\ \varphi\ \ \textbf{of} \\ &\quad \text{SAT} \Rightarrow \text{intp}^{\backprime}\ S.\text{runs} \subseteq \mathcal{L}\ \varphi\ | \\ &\quad \text{UNSAT}\ r \Rightarrow r \in S.\text{runs} \land \text{intp}\ r \notin \mathcal{L}\ \varphi \end{aligned} \tag{10}$$

For a system automaton 'S' and an LTL formula 'φ', the model checker returns 'SAT' if all runs of the system satisfy the formula, and '$\text{UNSAT}\ r$' if 'r' is a run that does not satisfy the

formula. Note that LTL formulae are independent of the system: They specify valid sequences of sets of atomic propositions, while a run of the system is a sequence of system states. The function 'intp' maps a run to the sequence of sets of atomic propositions that hold at the states of the run, and '$\mathrm{intp}^{\backprime}$' is its extension to sets of runs. Note that '$\mathrm{intp}^{\backprime}\ S.\mathrm{runs} = \mathcal{L}\ S$'.

Next, we assume that we have implementations of the components that match the implementations of the data. Formally, we fix *refinement relations* to relate the implementation of data to its abstract representation, and assume that the components' implementations *refine* the abstract components' specifications. This means that for related arguments, the implementation and the abstract component return related results. For example, we assume refinement relations '$\mathrm{gbg_rel}$' and '$\mathrm{resx_rel}$' for generalized Büchi graphs and results of the emptiness check, and an implementation '$\mathrm{echk_impl}$', such that:

$$(\mathrm{echk_impl}, \mathrm{echk_abs}) \in \mathrm{gbg_rel} \to \mathrm{resx_rel} \tag{11}$$

Then, we assemble the assumed implementations of the components to a model checker implementation '$\mathrm{cava_impl}$', and show that it refines the abstract model checker:

$$(\mathrm{cava_impl}, \mathrm{cava_abs}) \in \mathrm{sa_rel} \to \mathrm{res_rel} \tag{12}$$

Here, '$\mathrm{sa_rel}$' relates the system automata implementation to the abstract system automata, and '$\mathrm{res_rel}$' relates projected results.

Instantiating the above with actual implementations of the components and the data yields an executable model checker and its correctness theorem. This approach has several advantages. Firstly, the components of the model checker can be developed separately, as long as they match the implementations of the passed data. Secondly, the CAVA Automata Library [14] provides efficient implementations of the required automata types, which can be conveniently used by the components. Finally, components can easily be added or exchanged. For example, the current CAVA model checker supports two emptiness check algorithms, one based on nested DFS, and the other based on SCCs, which can be selected by a configuration option. Adding another algorithm amounts to proving that the algorithm refines the abstract emptiness check, and then adding a new configuration option to CAVA. In particular, it does *not* require changing other components or the overall correctness proof.

Finally, using this refinement-based approach allows for the seamless integration of many implementation techniques required to design an efficient model checker:

- When instantiating CAVA with a modeling language, the modeling language has a semantics which maps a model to an abstract system automaton. Moreover, we compile a model to a system automaton implementation, usually a successor function, which maps configurations to lists of successor configurations. Showing compiler correctness amounts to showing that the abstract system automaton is related to the concrete one. Then, the CAVA correctness theorem (theorem 12) implies that the result of the implementation is related to the abstract system automaton, that is, the semantics of the model.
- The states of the product automaton are constructed lazily. This saves memory if a counterexample is found before the whole state space is explored, as the unexplored parts of the state space do not occupy memory. This only affects the product construction component and the generalized Büchi graph that is implemented by its successor function.
- The alphabets of the automata are sets of atomic propositions. However, these sets have exponential size in general, so storing them explicitly is not efficient. Instead, the automaton constructed from the LTL formula stores two sets 'P' and 'N' of atomic propositions, representing all sets 'A' with '$P \subseteq A \wedge A \cap N = \{\}$'. This representation is naturally generated by many algorithms that convert LTL formulae to automata.

The automaton constructed from the model uses system states to represent the set of all atomic propositions that hold in a state. On product construction, it has to be decided whether the intersection of two sets of atomic propositions, one represented by '(P, N)' and the other represented by a system state 'S' is empty. This can simply be done by evaluating the atomic propositions in 'P' and 'N' on 'S'.
- The result of the emptiness check may contain a counterexample, which is an infinite run of the generalized Büchi graph. Clearly, a direct representation of infinite runs is not possible. However, a common representation is to use a *lasso*, that is, a finite path to an accepting state, and a finite, non-empty loop on this state, which has to contain states from all acceptance classes of the generalized Büchi graph. For a non-empty graph, there is always an accepting path that can be described by a lasso, which can be computed naturally by the emptiness check algorithms.

3.10 Integration of Partial Order Reduction

At this point, we have formalized all the necessary prerequisites and will now integrate partial order reduction into the Cava model checker. We refine the ample function, the execution function, and the interpretation function to efficiently executable implementations. This includes compilation of the model to a more efficient representation. Then, we replace the implementation of the successor function with the ample function. Instantiating the generic infrastructure of the Cava model checker then yields an executable LTL model checker 'cava_por' which uses the reduced system automaton. Combining its correctness theorem with that of abstract partial order reduction and the theorem about stuttering invariance of LTL properties then yields the main theorem of our development:

$$\textbf{case}\ \text{cava_por}\ S\ \varphi\ \textbf{of}\ \text{SAT} \Rightarrow \mathcal{L}\ S \subseteq \mathcal{L}\ \varphi \mid \text{UNSAT} \Rightarrow \mathcal{L}\ S \nsubseteq \mathcal{L}\ \varphi \tag{13}$$

This theorem states that the function 'cava_por' decides whether or not the sequences of atomic propositions admitted by runs of the program satisfy the LTL formula. The meaning of this statement only depends on the abstract semantics of the SM language ($\mathcal{L}\ S$) and the abstract semantics of LTL formulae ($\mathcal{L}\ \varphi$). All other parts of the formalization, including partial order reduction, LTL model checking, and implementations, are covered by this machine-checked correctness theorem. Note that the model checker can actually provide a counterexample in case the program does not satisfy the formula. However, we only show the simplified view here as it is easier to understand.

Finally, Isabelle/HOL can generate Standard ML code from the definition of the function 'cava_por'. This code then constitutes a formally verified and executable LTL model checker. A snapshot of this formalization can be found at `https://cava.in.tum.de/CAVA_POR`. We are currently working on integrating the partial order reduction formalization into an up-to-date AFP entry of the Cava model checker, which can be found at `https://www.isa-afp.org`.

We conclude with some statistics about the formalization, which took about 15 person-months and resulted in about 13k lines of theory text being added to the model checker. This includes both definitions and proofs and splits up into 6k lines for abstract partial order reduction and 7k lines for the SM language and the associated program analysis. The size of the whole codebase of the model checker and its libraries is about 140k lines of theory text.

4 Evaluation

We perform some basic sanity checks using systems that admit no reduction and complete sequentialization. As a practical example, we implement a distributed mutual exclusion algorithm called MULOG [24] using the supported PROMELA fragment. The tested property specifies that at most one process can be in the critical section at any point in time. We perform model checking using both the CAVA and the SPIN model checkers, both with and without partial order reduction. Figure 3 shows the reduction effectiveness for this algorithm.

n	SPIN	SPIN*	Factor SPIN	CAVA	CAVA*	Factor CAVA
1	27	27	1	52	52	1
2	2,674	2,004	1.33	5,538	4,284	1.29
3	2,376,180	1,171,578	2.03	5,205,376	2,779,218	1.87

Fig. 3: Reduction Effectiveness for MULOG. Shown are the number of states that were explored during model checking using both the SPIN and the CAVA model checkers for a given number of processes 'n'. The starred variants indicate where partial order reduction was used. The table also shows the reduction factor that was achieved by each model checker.

Both the CAVA and the SPIN model checker show a significant reduction in the number of states. The reduction factors are comparable (roughly 1.3 for '$n = 2$' and roughly 2 for '$n = 3$'). The SPIN model checker explores fewer states in total (roughly factor 2) and has shorter execution times (roughly factor 400) than the CAVA model checker.

We would like to emphasize that in this paper, it is not our goal to compete with SPIN. Instead, our focus is on providing a verified and executable reference implementation of partial order reduction. The SPIN model checker employs various other optimizations and compilation to C code, while the CAVA model checker interprets the semantics of the modeling language. Thus, little insight can be gained by directly comparing execution time and memory consumption. Incorporating these optimizations is orthogonal to partial order reduction and we consider this subject of further research. Due to the modular architecture of the CAVA model checker, doing so will not make this contribution obsolete. At this point, it will also be possible to perform a more comprehensive evaluation with multiple example algorithms.

5 Dynamic Partial Order Reduction with On-The-Fly Model Checking

This section presents a counterexample for the invalid lemma mentioned in section 2.2. In [28], the partial order reduction algorithm designed for off-line model checking is modified and used with on-the-fly model checking. When doing off-line partial order reduction, the reduced system automaton is explored via depth-first search and its product with the formula automaton '$\mathcal{B}$' is checked for emptiness. On-the-fly partial order reduction consists of defining the reduced product automaton '$\mathcal{A}'$' directly and checking its emptiness while exploring it via nested depth-first search.

The correctness proof introduces an intermediate automaton 'G'' that is structurally similar to '$\mathcal{A}'$' but fulfills the conditions of the off-line correctness theorems. Together with the formula automaton '$\mathcal{B}$', the following claim is then used to prove correctness:

$$\mathcal{L}\ \mathcal{A}' = \mathcal{L}\ G' \cap \mathcal{L}\ \mathcal{B} \tag{14}$$

However, in this section, we will present an example for which this claim does not hold. This example is adapted from [3, section 8.4], where we originally discovered the problem.

Figures 4 and 5 show the system automaton 'G' and the formula automaton '$\mathcal{B}$', respectively.

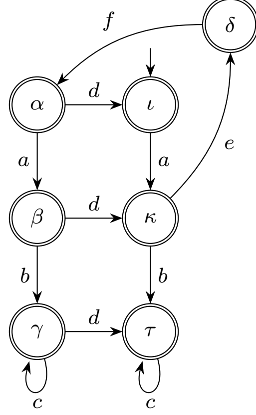


Fig. 4: System Automaton 'G'.

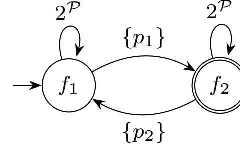


Fig. 5: Formula Automaton '$\mathcal{B}$'.

We define the interpretation function 'intp' as follows:

$$\mathrm{intp}\ q = \begin{cases} \{p_1\} & \text{if } q = \delta \\ \{p_2\} & \text{if } q = \gamma \\ \{p_2\} & \text{if } q = \tau \\ \{\} & \text{otherwise} \end{cases} \tag{15}$$

With this, we have '$\mathrm{visible} = \{b, e, f\}$'.

We use the independence relation '$I = \{(a, d), (d, a), (b, d), (d, b), (c, d), (d, c)\}$' and define the ample function as follows:

$$\mathrm{ample}\ (x, y) = \begin{cases} \{d\} & \text{if } x = \alpha \text{ and } (\iota, y) \text{ is not open} \\ \{a\} & \text{if } x = \alpha \text{ and } (\beta, y) \text{ is not open} \\ \mathrm{en}\ x & \text{otherwise} \end{cases} \tag{16}$$

If more than one condition is met, the topmost valid equation is used. This function fulfills all the necessary conditions.

Figure 6 shows the reduced product automaton '$\mathcal{A}'$' generated by the ample function given above. In this case, the intermediate automaton 'G'' looks exactly like '$\mathcal{A}'$', except that all states are accepting.

In order to create a counterexample, we define the word 'w':

$$w = \{\} \cdot \{\} \cdot \{p_1\} \cdot \{\} \cdot \{\} \cdot \{p_2\}^{\omega} \tag{17}$$

We have '$w \in \mathcal{L}\ G' \wedge w \in \mathcal{L}\ \mathcal{B} \wedge w \notin \mathcal{L}\ \mathcal{A}'$' and thus obtain:

$$\mathcal{L}\ \mathcal{A}' \neq \mathcal{L}\ G' \cap \mathcal{L}\ \mathcal{B} \tag{18}$$

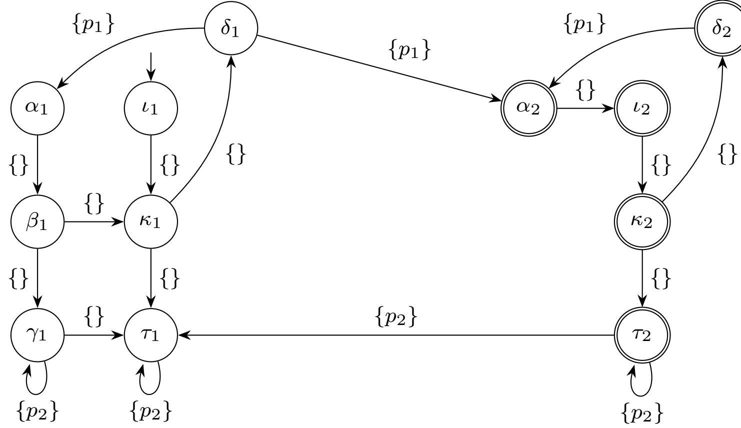


Fig. 6: Reduced product automaton '$\mathcal{A}'$'. We use shorthand state notation. For instance, the state '(α, f_1)' is simply written as 'α_1'.

This contradicts the claim 14 made earlier. However, the proof only needs a weaker version of the claim:

$$\mathcal{L}\ \mathcal{A}' = \{\} \iff \mathcal{L}\ G' \cap \mathcal{L}\ \mathcal{B} = \{\} \tag{19}$$

Our example does not contradict this statement and we do in fact believe that it holds true. Unfortunately, we have not been able to find a proof for this statement.

6 Conclusion

Formal verification is sometimes downplayed as "careful documentation of proven theorems" or "filling in obvious details in proofs". In practice, formal proofs usually involve extensive modeling as well as abstraction, generalization, and simplification. What may seem like trivial completion of the informal proof often involves bridging large gaps and proving omitted corner cases. In this project, it even helped us discover an issue with the correctness proof given in [28]. This demonstrates both the need for and the usefulness of formal verification.

More importantly, we developed a formally verified and executable LTL model checker with partial order reduction. As the verification is machine-checked and covers everything from the abstract algorithm to the generated SML code, this is a very strong correctness guarantee. Our model checker is fast enough to serve as a reference implementation for other model checkers on models of realistic size. This constitutes a much-needed source of trust given the widespread use of partial order reduction together with its history of issues. The formalization can further serve as a detailed description of the theory of partial order reduction and its correctness proof, which is useful since nontrivial gaps were bridged in the proof. We also developed a significant amount of foundational theories that can be reused in other projects. Finally, our work demonstrates that large systems can now be verified using proof assistants via modularization and reuse of existing theories.

Future work consists of extending the SM language to make it more practical, with the ultimate goal of supporting most or all of the features of PROMELA. It is also possible to find smaller sets of sticky actions by incorporating heuristics about variable increments/decrements [10]. Another way to improve reduction consists of using additional static analysis to find

larger independence relations. Finally, there is still room for improvement concerning the implementation, especially via the use of imperative data structures [13].

References

[1] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus. A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998.

[2] Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. “Truly Modular (Co)datatypes for Isabelle/HOL”. In: *ITP*. Vol. 8558. LNCS. Springer, 2014, pp. 93–110.

[3] Julian Brunner. “Implementation and Verification of Partial Order Reduction for On-The-Fly Model Checking”. MA thesis. Technische Universität München, July 15, 2014. 83 pp. URL: `http://www21.in.tum.de/~brunnerj/documents/ivporotfmc.pdf`.

[4] Julian Brunner and Peter Lammich. “Formal Verification of an Executable LTL Model Checker with Partial Order Reduction”. In: *NFM*. Springer, 2016, pp. 307–321.

[5] Ching-Tsun Chou and Doron Peled. “Formal Verification of a Partial-Order Reduction Technique for Model Checking”. In: *TACAS*. Vol. 1055. LNCS. Springer, 1996, pp. 241–257.

[6] Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. “A Fully Verified Executable LTL Model Checker”. In: *CAV*. Vol. 8044. LNCS. Springer, 2013, pp. 463–478.

[7] Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. “A Fully Verified Executable LTL Model Checker”. In: *Archive of Formal Proofs* (May 2014). Formal proof development. URL: `http://afp.sf.net/entries/CAVA_LTL_Modelchecker.shtml`.

[8] Gerard J. Holzmann. *The SPIN Model Checker. Primer and Reference Manual*. Addison-Wesley Professional, Sept. 2003.

[9] Gerard J. Holzmann, Doron Peled, and Mihalis Yannakakis. “On Nested Depth First Search”. In: *SPIN Workshop*. Vol. 32. 1996, pp. 81–89.

[10] Robert Kurshan, Vladimir Levin, Marius Minea, Doron Peled, and Hüsnü Yenigün. “Static Partial Order Reduction”. In: *TACAS*. Vol. 1384. LNCS. Springer, 1998, pp. 345–357.

[11] Peter Lammich. “Collections Framework”. In: *Archive of Formal Proofs* (Nov. 2009). Formal proof development. URL: `http://afp.sf.net/entries/Collections.shtml`.

[12] Peter Lammich. “Refinement for Monadic Programs”. In: *Archive of Formal Proofs* (Jan. 2012). Formal proof development. URL: `http://afp.sf.net/entries/Refine_Monadic.shtml`.

[13] Peter Lammich. “Refinement to Imperative/HOL”. In: *ITP*. Vol. 9236. LNCS. Springer, 2015, pp. 253–269.

[14] Peter Lammich. “The CAVA Automata Library”. In: *Archive of Formal Proofs* (May 2014). Formal proof development. URL: `http://afp.sf.net/entries/CAVA_Automata.shtml`.

[15] Peter Lammich. “The Imperative Refinement Framework”. In: *Archive of Formal Proofs* (Aug. 2016). `http://isa-afp.org/entries/Refine_Imperative_HOL.shtml`, Formal proof development. ISSN: 2150-914x.

[16] Peter Lammich. “Verified Efficient Implementation of Gabow’s Strongly Connected Component Algorithm”. In: *ITP*. Vol. 8558. LNCS. Springer, 2014, pp. 325–340.

[17] Peter Lammich and Andreas Lochbihler. “The Isabelle Collections Framework”. In: *ITP*. Vol. 6172. LNCS. Springer, 2010, pp. 339–354.

[18] Peter Lammich and René Neumann. “A Framework for Verifying Depth-First Search Algorithms”. In: *CPP*. ACM, Jan. 13, 2015, pp. 137–146.

[19] Peter Lammich and Thomas Tuerk. “Applying Data Refinement for Monadic Programs to Hopcroft’s Algorithm”. In: *ITP*. Vol. 7406. LNCS. Springer, 2012, pp. 166–182.

[20] Peter Lammisch. “The CAVA Automata Library”. In: *Isabelle Workshop 2014*. May 2014.

[21] Andreas Lochbihler. “Coinductive”. In: *Archive of Formal Proofs* (Feb. 2010). Formal proof development. URL: `http://afp.sf.net/entries/Coinductive.shtml`.

[22] Antoni Mazurkiewicz. “Trace Theory”. In: *Advances in Petri Nets, Part II*. Vol. 255. LNCS. Springer, 1987, pp. 278–324.

[23] Stephan Merz. “Stuttering Equivalence”. In: *Archive of Formal Proofs* (May 2012). Formal proof development. URL: `http://afp.sf.net/entries/Stuttering_Equivalence.shtml`.

[24] Mohamed Naimi, Michel Trehel, and André Arnold. “A Log (N) Distributed Mutual Exclusion Algorithm Based on Path Reversal”. In: *Journal of Parallel and Distributed Computing* 34.1 (1996), pp. 1–13.

[25] René Neumann. “Using Promela in a Fully Verified Executable LTL Model Checker”. In: *VSTTE*. LNCS. Springer, 2014, pp. 105–114.

[26] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. Vol. 2283. LNCS. Springer, 2002.

[27] Larry Paulson, Tobias Nipkow, and Makarius Wenzel. *Isabelle*. 2014. URL: `http://isabelle.in.tum.de`.

[28] Doron Peled. “Combining Partial Order Reductions with On-the-Fly Model-Checking”. In: *Formal Methods in System Design* 8.1 (1996), pp. 39–64.

[29] Doron Peled and Thomas Wilke. “Stutter-Invariant Temporal Properties are Expressible Without the Next-Time Operator”. In: *Information Processing Letters* 63.5 (1997), pp. 243–246.

[30] Philip Wadler. “Comprehending Monads”. In: *Mathematical Structures in Computer Science* 2 (04 Dec. 1992), pp. 461–493.

Appendix B

Büchi Complementation

This appendix includes a full copy of the following publication.

Julian Brunner. “Formal Verification of Executable Complementation and Equivalence Checking for Büchi Automata”. In: *Integrated Formal Methods - 16th International Conference, IFM 2020, Lugano, Switzerland, November 16-20, 2020, Proceedings*. Ed. by Brijesh Dongol and Elena Troubitsyna. Vol. 12546. Lecture Notes in Computer Science. Springer, 2020, pp. 239–256. DOI: 10.1007/978-3-030-63461-2_13

This article is included as part of this publication-based dissertation.

Copyright

The proceedings article in this appendix was published by Springer Nature. Per author retained rights, it can be reused in dissertations without obtaining permission from the publisher. The accepted version of the manuscript is thus reproduced here in accordance with the requirements of the publisher.

Summary

This article describes the formal verification of algorithms for Büchi complementation and equivalence checking using the proof assistant Isabelle. We follow the rank-based approach to Büchi complementation [KV01]. This method introduces the concept of the run DAG, which contains all of the possible paths that an automaton can take while reading a word. Run DAGs are then used to define odd rankings for a given word, which exist if and only if the automaton rejects said word. Finally, the complement automaton is designed to nondeterministically search for such an odd ranking, accepting if and only if one exists.

We make heavy use of the library for transition systems and automata, which was originally developed to enable the formalization of Büchi complementation. Using this library, we define the concepts of run DAGs, odd rankings, as well as the actual complement automata. Based on these definitions, we formalize the correctness proof that is given in [KV01]. This includes the equivalence between rejection by the original automaton and odd rankings, as well as the one between odd rankings and the acceptance by the complement automaton. We then use refinement to derive an executable implementation for this complementation algorithm.

We extend the library for transition systems and automata with intersection and union operations. In conjunction with an emptiness check on Büchi automata, this allows us to use Büchi complementation to define decision procedures for language containment and equality. Again, we use refinement to derive executable implementations for these algorithms.

We integrate both the complementation algorithm and the decision procedure for language-wise equivalence into a command-line tool. This tool can then be used to process Büchi automata in the Hanoi Omega-Automata format. It represents a verified reference implementation that can be used to check unverified tools. As all aspects are integrated into one formalization, the verification covers everything from the abstract correctness proofs down to the generated SML code. Finally, we evaluate the performance of our tool by comparing it against Spot and GOAL as well as using it to check LTL translation algorithms from Spot and Owl.

Contributions

I am the only author of this article. Thus, all contributions are mine.

Formal Verification of Executable Complementation and Equivalence Checking for Büchi Automata*

Julian Brunner[0000-0001-8922-6097]

Technische Universität München brunnerj@in.tum.de

Abstract. We develop a complementation procedure and an equivalence checker for nondeterministic Büchi automata. Both are formally verified using the proof assistant Isabelle/HOL. The verification covers everything from the abstract correctness proof down to the generated SML code.
The complementation follows the rank-based approach. We formalize the abstract algorithm and use refinement to derive an executable implementation. In conjunction with a product operation and an emptiness check, this enables deciding language-wise equivalence between nondeterministic Büchi automata. We also improve and extend our library for transition systems and automata presented in previous research.
Finally, we develop a command-line executable providing complementation and equivalence checking as a verified reference tool. It can be used to test the output of other, unverified tools. We also include some tests that demonstrate that its performance is sufficient to do this in practice.

Keywords: formal verification · omega automata · complementation

1 Introduction

Büchi complementation is the process of taking a Büchi automaton, and constructing another Büchi automaton which accepts the complementary language. It is a much-researched topic [10, 37, 20, 15, 38]. In fact, it has been so popular that there are now several meta-papers [43, 41] chronologuing the research itself. Much of this research has focused on the state complexity of the resulting automata (see section 2.3). However, Büchi complementation also has compelling applications. Model checking usually requires having the property to be checked against as either a formula or a deterministic Büchi automaton, as those are easily negated and complemented, respectively [15]. However, having access to a general complementation procedure, it becomes possible to decide language containment between arbitrary nondeterministic Büchi automata. This not only allows for more general model checking, but also enables checking if two automata are equivalent in terms of their language.

* Research supported by DFG grant CAVA (Computer Aided Verification of Automata, ES 139/5-1, NI 491/12-1, SM 73/2-1) and CAVA2 (Verified Model Checkers, KR 4890/1-1, LA 3292/1-1)

Unfortunately, complementation algorithms are complicated and their correctness proofs are involved. This is common in the model checking setting and there are examples of algorithms widely believed to be correct turning out not to be [18, 8, 40]. The situation is especially troubling as these tools act as trust multipliers. That is, the trust in the correctness of one tool is used to justify confidence in the correctness of the many entities that it checks. Motivated by this situation, our goal is to formally verify one such complementation algorithm.

We use the proof assistant Isabelle/HOL [34] for this. Thanks to its LCF-like architecture, Isabelle/HOL and the formalizations it facilitates grant very strong correctness guarantees. Our contributions are as follows.

1. Formalization of rank-based complementation [20] theory
2. Formally verified complementation implementation
3. Formally verified equivalence checker
4. Extension and continued development of automata library [7, 9]

In previous work, Stephan Merz formalized complementation of weak alternating automata [33]. He also started working on a formalization of Büchi complementation. However, this only covers the first part of the complementation procedure and was never finished or published. Thus, our work constitutes what we believe to be the first formally verified implementation of Büchi complementation. The verification gaplessly covers everything from the abstract correctness proof to the executable SML code.

The equivalence checker can be used as a command-line tool to check automata in the Hanoi Omega-Automata format [1]. It takes the role of a trusted reference implementation that can be used to test the correctness of other tools, like those translating from LTL formulae to automata. With an equivalence checker, it is possible to test whether several algorithms produce automata with identical languages, given the same formula. It is also possible to test if an algorithm that simplifies automata preserves their language.

2 Theory

We follow the rank-based complementation construction described in [20]. The central concept here is the *odd ranking*, a function f that certifies the rejection of a word w by the automaton A. The complement automaton $\overline{A}$ is then designed to nondeterministically search for such an odd ranking, accepting if and only if one exists. Thus, the complement automaton accepts exactly those words that the original automaton rejects.

$$w \notin \mathcal{L}\ A \iff \exists f.\ \text{odd_ranking}\ A\ w\ f \iff w \in \mathcal{L}\ \overline{A} \tag{1}$$

Having access to complement and product operations as well as an emptiness check, we can then decide language containment.

$$\mathcal{L}\ A \subseteq \mathcal{L}\ B \iff \mathcal{L}\ A \cap \overline{\mathcal{L}\ B} = \emptyset \iff \mathcal{L}\ \left(A \times \overline{B}\right) = \emptyset \tag{2}$$

Checking for containment in both directions then leads to a decision procedure for language-wise equivalence of Büchi automata.

2.1 Notation

We introduce some basic notation. Let $w \in \Sigma^\omega$ be an infinite sequence and $w_k \in \Sigma$ be the symbol at index k in w. Let $A = (\Sigma, Q, I, \delta, F)$ be a nondeterministic Büchi automaton with alphabet Σ, states Q, initial states $I :: Q$ set, successor function $\delta :: \Sigma \to Q \to Q$ set, and acceptance condition $F :: Q \to$ bool. Let $\mathcal{L}\ A \subseteq \Sigma^\omega$ denote the language of automaton A.

2.2 Complementation

We want to realize complementation according to equation 1. For this, we need to define odd rankings and the complement automaton. We also need to define run DAGs as a prerequisite for odd rankings.

A run DAG is a graph whose nodes are pairs of states and natural numbers. Given an automaton A and a word w, we define it inductively as follows.

Definition 1 (Run DAG). $G = (V, E)$ *with* $V \subseteq Q \times \mathbb{N}$ *and* $E \subseteq V \times V$

$$\begin{aligned} & p \in I && \Longrightarrow && (p, 0) \in V \\ (p, k) \in V \Longrightarrow\ & q \in \delta\ w_k\ p && \Longrightarrow && (q, k+1) \in V \\ (p, k) \in V \Longrightarrow\ & q \in \delta\ w_k\ p && \Longrightarrow && ((p, k), (q, k+1)) \in E \end{aligned}$$

Intuitively, each node $(p, k) \in V$ represents A being in state p after having read k characters from w. With that, the run DAG contains all possible paths that A can take while reading w.

We can now define odd rankings. An odd ranking is a function assigning a rank to each node in the run DAG. Given an automaton A and a word w, we require the following properties to hold for odd rankings.

Definition 2 (Odd Ranking). odd_ranking $A\ w\ f$ *with* $f :: V \to \mathbb{N}$

$$\begin{aligned} &\forall\, v \in V. && f\ v \leq 2\,|Q| \\ &\forall\, (u, v) \in E. && f\ u \geq f\ v \\ &\forall\, (p, k) \in V. && F\ p \Longrightarrow \text{even}\ (f\ (p, k)) \\ &\forall\, r \in \text{paths}\ G. && \textit{the path } r \textit{ eventually gets stuck in an odd rank} \end{aligned}$$

Intuitively, the rank of a node indicates the distance to a node from which no more accepting states are visited [15].

The final definition concerns the actual complement automaton. Given an automaton $A = (\Sigma, Q, I, \delta, F)$, we define its complement as follows.

Definition 3 (Complement Automaton). $\overline{A} = (\Sigma, Q_C, I_C, \delta_C, F_C)$

$$
\begin{array}{rcl}
\delta_1 & :: & \Sigma \to (Q \rightharpoonup \mathbb{N}) \to (Q \rightharpoonup \mathbb{N}) \text{ set} \\
g \in \delta_1 \; a \; f & \Longleftrightarrow & \text{dom } g = \bigcup p \in \text{dom } f.\, \delta \; a \; p \; \wedge \\
 & & \forall p \in \text{dom } f.\, \forall q \in \delta \; a \; p.\, f \; p \geq g \; q \; \wedge \\
 & & \forall q \in \text{dom } g.\, F \; q \implies \text{even } (g \; q)
\end{array}
$$

$$
\begin{array}{rcl}
\delta_2 & :: & \Sigma \to (Q \rightharpoonup \mathbb{N}) \to Q \text{ set} \to Q \text{ set} \\
\delta_2 \; a \; g \; P & = & \begin{cases} \{q \in \text{dom } g \,|\, \text{even } (g \; q)\} & \textit{if } P = \{\} \\ \{q \in \bigcup p \in P.\, \delta \; a \; p \,|\, \text{even } (g \; q)\} & \textit{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{rcl}
Q_C & :: & ((Q \rightharpoonup \mathbb{N}) \times Q \text{ set}) \text{ set} \\
Q_C & = & \delta_C^* \; \Sigma \; I_C
\end{array}
$$

$$
\begin{array}{rcl}
I_C & :: & Q_C \text{ set} \\
I_C & = & (\lambda \, p \in I.\, 2 \, |Q| \, , \emptyset)
\end{array}
$$

$$
\begin{array}{rcl}
\delta_C & :: & \Sigma \to Q_C \to Q_C \text{ set} \\
\delta_C \; a \; (f, P) & = & \{(g, \delta_2 \; a \; g \; P) \,|\, g \in \delta_1 \; a \; f\}
\end{array}
$$

$$
\begin{array}{rcl}
F_C & :: & Q_C \to \text{bool} \\
F_C \; (f, P) & = & (P = \emptyset)
\end{array}
$$

Since the complement automaton is designed to nondeterministically search for an odd ranking, many of the properties from definition 2 reappear here. Instead of a ranking on the whole run DAG ($V \to \mathbb{N}$), the complement automaton deals with *level rankings*. These assign ranks to only the reachable nodes in the current level ($Q \rightharpoonup \mathbb{N}$). Furthermore, each state keeps track of which paths have yet to visit an odd rank (Q set). This encodes the property of every path getting stuck in an odd rank, with the acceptance condition requiring this set to become empty infinitely often. Together, these lead to the state type $(Q \rightharpoonup \mathbb{N}) \times Q$ set.

2.3 Complexity and Optimizations

Much of the interest in Büchi complementation focuses on its state complexity [43, 15, 38]. That is, one considers the number of states in the complement automaton as a function of the number of states in the original automaton. For an automaton with n states, the original construction by Büchi [10] resulted in $2^{2^{\mathcal{O}(n)}}$ states [15]. The complementation procedure derived from Safra's determinization construction [37] reduces this to $2^{\mathcal{O}(n \log n)}$ or n^{2n} states [15]. The algorithm from [20] generates a complement automaton with at most $(6n)^n$ states [15]. In the quest for closing the gap between the known lower and upper bounds, various optimizations to this algorithm have been proposed. The optimization in [15] lowers the bound to $\mathcal{O}((0.96n)^n)$ states. The algorithm is then adjusted further in [38] to lower the bound to $\mathcal{O}((0.76n)^n)$ states.

This being the first attempt at formalizing Büchi complementation, we chose not to implement these more involved optimizations. Instead, we favor the original version of the algorithm as presented in [20]. We do however implement one optimization mentioned in [20]. In definition 3, for each successor q of p, the function δ_1 considers for q all ranks lower than or equal to the rank of p. We restrict δ_1 so that it only considers for q a rank that is equal to or one less than the rank of p. This does not change the language of the complement automaton and significantly restricts the number of successors generated for each state.

It is worth noting that in practice, factors other than asymptotical state complexity can also play a role. For instance, it turns out that determinization-based complementation often generates fewer states than rank-based complementation [41]. This is despite the fact that rank-based complementation is optimal in terms of asymptotical state complexity.

2.4 Equivalence

We want to realize equivalence according to equation 2. For this, we need to define a product operation and an emptiness check on Büchi automata.

The product construction follows the textbook approach, where the product of two nondeterministic Büchi automata results in one nondeterministic generalized Büchi automaton. For the emptiness check, we use Gabow's algorithm for strongly-connected components [16]. This enables checking emptiness of generalized Büchi automata directly, skipping the degeneralization to regular Büchi automata that is usually necessary for nested-DFS-based algorithms.

3 Formalization

With the theoretical background established, we now describe the various aspects of our formalization. This section will mostly give a high-level overview, highlighting challenges and points of interest while avoiding technical details. However, specific parts of the formalization will be presented in greater detail.

3.1 Isabelle/HOL

Isabelle/HOL [34] is a proof assistant based on Higher-Order Logic (HOL), which can be thought of as a combination of functional programming and logic. Formalizations done in Isabelle are trustworthy due to its LCF architecture. It guarantees that all proofs are checked using a small logical core which is rarely modified but tested extensively over time, reducing the trusted code base to a minimum.

Code generation in Isabelle/HOL is based on a shallow embedding of HOL constants in the target language. Equational theorems marked as code equations are translated into rewrite rules in the target language [17]. This correspondence embodies the specification of the target language semantics. As this process does not involve the LCF kernel, the code generator is part of the trusted code base.

3.2 Basics

The most basic concept needed for our formalization is that of sequences. The HOL standard library already includes extensive support for both finite and infinite sequences. They take the form of the types list and stream.

Definition 4 (Sequences).

$$\textbf{datatype}\ \alpha\ \text{list} = [\,]\ |\ \alpha \mathbin{\#} \alpha\ \text{list}$$
$$\textbf{codatatype}\ \alpha\ \text{stream} = \alpha \mathbin{\#\#} \alpha\ \text{stream}$$

The new datatype package [4, 3] allows for codatatypes like stream. The libraries of both list and stream include many common operations and their properties.

We also make use of a shallow embedding of linear temporal logic (LTL) on streams that is defined using inductive and coinductive predicates. This is used to define a predicate holding infinitely often in an infinite sequence.

Definition 5 (Infinite Occurrence). $\text{infs}\ P\ w \iff \text{alw}\ (\text{ev}\ (\text{holds}\ P))\ w$

3.3 Transition Systems and Automata

In our formalization, we both use and extend the *Transition Systems and Automata* library [7, 9]. The development of this library was in fact motivated by the idea of formalizing Büchi complementation and determinization. Since then, it has been used in several other formalizations [6, 8, 39, 9, 35, 36].

The goal of this library is to support many different types of automata while avoiding both duplication and compromising usability. This is achieved via several layers of abstraction as well as the use of Isabelle's locale mechanism. For an in-depth description, see [9]. Since then, an additional abstraction layer has been introduced to consolidate various operations on automata like intersection, union, and degeneralization. However, describing this in detail is outside the scope of this paper. Thus, we will only introduce the concepts and constants that are used in later sections. We start with the definition of a transition system.

Definition 6 (Transition System).

$$\begin{array}{l}\textbf{locale}\ \text{transition_system} = \\ \quad \textbf{fixes}\ \text{execute} :: \textit{transition} \Rightarrow \textit{state} \Rightarrow \textit{state} \\ \quad \textbf{fixes}\ \text{enabled} :: \textit{transition} \Rightarrow \textit{state} \Rightarrow \text{bool}\end{array}$$

It fixes type variables for transitions and states as well as constants to determine which transitions are enabled in each state and which target states they lead to. This locale forms the backbone of the library. Note that it may look like it can only be used to model (sub-)deterministic transition systems. However, by instantiating the type variable *transition*, we can actually model many different types of transition systems, including nondeterministic ones [9].

We can then define concepts concerning sequences of transitions.

Definition 7 (Targets and Traces).

$$\text{target} = \text{fold execute} :: \textit{transition} \text{ list} \Rightarrow \textit{state} \Rightarrow \textit{state}$$
$$\text{trace} = \text{scan execute} :: \textit{transition} \text{ list} \Rightarrow \textit{state} \Rightarrow \textit{state} \text{ list}$$
$$\text{strace} = \text{sscan execute} :: \textit{transition} \text{ stream} \Rightarrow \textit{state} \Rightarrow \textit{state} \text{ stream}$$

Given a sequence of transitions and a source state, these functions give the target state and the finite and infinite sequence of traversed states, respectively. Note how each of these is simply a lifted version of execute.

We can also define constants for finite and infinite paths, respectively.

Definition 8 (Paths).

$$\textbf{inductive } \text{path} :: \textit{transition} \text{ list} \Rightarrow \textit{state} \Rightarrow \text{bool } \textbf{where}$$
$$\text{path } [] \; p$$
$$\text{enabled } a \; p \implies \text{path } r \; (\text{execute } a \; p) \implies \text{path } (a \# r) \; p$$
$$\textbf{coinductive } \text{spath} :: \textit{transition} \text{ stream} \Rightarrow \textit{state} \Rightarrow \text{bool } \textbf{where}$$
$$\text{enabled } a \; p \implies \text{spath } r \; (\text{execute } a \; p) \implies \text{spath } (a \#\# r) \; p$$

These constants are (co)inductively defined predicates that capture the notion of all the transitions in a sequence being enabled at their respective states. Like before, these are lifted versions of enabled, which is also reflected in their types.

3.4 Run Dags

Having established all the basics and foundations, we can now turn to the actual formalization of Büchi complementation. We start with formalizing definition 1 concerning run Dags. We do this by instantiating the transition system locale from definition 6. This yields definitions for all the required graph-related concepts, like finite and infinite paths as well as reachability.

We then establish a tight correspondence between these definitions and the ones concerning automata. This requires mostly elemental induction and coinduction proofs. Only minor technical work was required to translate between paths in the automaton being labeled and paths in its run Dag being indexed.

3.5 Odd Rankings

Having formalized run Dags, we can now formalize definition 2 concerning odd rankings. The resulting formal definition does not differ significantly from its informal counterpart and will thus not be repeated here.

We prove the left equivalence from equation 1, which states that an odd ranking f exists if and only if the automaton A rejects the word w.

$$w \notin \mathcal{L} \; A \iff \exists f. \; \text{odd_ranking } A \; w \; f$$

We follow the proof given in [20].

The direction $\Longleftarrow$ is fairly straightforward. Given an odd ranking, we immediately have that all infinite paths in the run DAG get trapped in an odd rank. Together with the fact that odd ranks are not accepting, we obtain that all infinite paths in the automaton are not accepting. Formally proving this is mainly technical work consisting of establishing the correspondence between the run DAG and the automaton. However, there is one exception. In [20], the fact that all infinite paths get trapped in some rank is merely stated as part of the definition of rankings. While this is intuitively obvious from the fact that ranks are natural numbers and always decreasing along a path, it still requires rigorous proof in a formal setting. Thus, we need to define the notion of decreasing infinite sequences and prove this property via well-founded induction on the ranks.

The direction $\Longrightarrow$ is a lot more involved. It requires defining an infinite sequence of subgraphs of the run DAG in order to construct an odd ranking. Again, we follow the proof given in [20]. As before, we were able to follow the high-level ideas of this proof in the formalized version, with some parts requiring more fine-grained reasoning or additional technical work. However, we want to highlight one particular technique that is used several times in the proof and that required special attention in the formalized version. While most of our descriptions focus on high-level ideas, we also want to take this opportunity to present one part of the formalization in greater detail.

The idea in question concerns itself with the construction of infinite paths in graphs and transition systems. We already encountered this type of reasoning in [8]. Assume that there is a state with property P, and that for every state with property P we can find a path to another state with property P. Then, there exists an infinite path that contains infinitely many states with property P. Intuitively, this seems obvious, which is why in informal proofs, statements like these require no further elaboration. However, in a formal setting, this requires rigorous reasoning, resulting in the following proof.

Lemma 1 (Recurring Condition).

```
lemma recurring_condition:
  assumes "P p" "∀p. P p ⟹ ∃r. r ≠ [] ∧ path r p ∧ P (target r p)"
  obtains r where "spath r p" "infs P (p ## strace r p)"
proof -
  obtain f where "f p ≠ []" "path (f p) p" "P (target (f p) p)"
    if "P p" for p by ...
  let ?g = "λp. target (f p) p"
  let ?r = "λp. flat (smap f (siterate ?g p))"
  have "?r p = f p @- ?r (?g p)" if "P p" for p by ...
  show ?thesis
  proof
    show "spath (?r p) p" by ...
    show "infs P (p ## strace (?r p) p)" by ...
  qed
qed
```

This theorem is stated for general transition systems and corresponds closely to the informal one presented earlier. That is, we assume that P holds at some state p. We also assume that for every state p where P holds, we can find a nonempty path r leading to a state target r p where P holds again. We prove that from these assumptions, one can obtain an infinite path r such that for the states it traverses, P holds infinitely often. The proof consists of three major steps.

1. Skolemization of the assumption. We obtain a function f that for each state in which P holds, gives a nonempty path that leads to another state in which P holds. This can be done by either explicitly invoking the choice theorems derived from Hilbert's epsilon operator, or by using `metis`.
2. Definition of the state iteration function $?g$ and the infinite path $?r$. We define a function $?g$ that for each state p gives the target state of the path given by f. Iterating $?g$ yields all those states along the infinite path where P holds. We can then define $?r$, which is the infinite path obtained by concatenating all the finite paths given by f from each state in the iteration of $?g$.
3. Proving the required properties of $?r$. We now prove both that $?r$ is an infinite path and that for the states it traverses, P holds infinitely often. Both of these proofs require specific coinduction rules for the constants spath and infs. This is because the coinduction cannot consume the infinite sequence one item at a time, instead having to operate on finite nonempty prefixes. However, these coinduction rules are generally useful and once proven, can be reused and improve compositionality.

In the end, a surprising amount of work is necessary to prove a seemingly obvious statement. While it is easy to dismiss this as a shortcoming of either formal logic in general or of a particular proof assistant, we do not think that this is the case. Instead, we believe that situations like these point out areas where informal proofs rely on intuition, thereby hiding the actual complexity of the proof. Since this can lead to subtle mistakes, the ability of formal proofs to make it visible is valuable and one of the reasons for our confidence in them. It is also worth noting that these situations do not persistently hinder the construction of formal proofs. By proving the statement in its most general form, this needs to be done only once for this type of reasoning to become available everywhere.

3.6 Complement Automaton

Next, we formalize definition 3 concerning the complement automaton. As in the previous section, the resulting formal definition differs only slightly from the informal one and will thus not be repeated here.

We prove the right equivalence from equation 1, which states that the complement automaton $\overline{A}$ accepts a word w if and only if an odd ranking f exists.

$$\exists f.\ \text{odd_ranking}\ A\ w\ f \iff w \in \mathcal{L}\ \overline{A} \tag{3}$$

We follow the proof given in [20].

There are two main challenges to formalizing this proof. The first one is converting between different representations of rankings. On the side of the odd ranking, a ranking is a function assigning ranks to the nodes in the run Dag. On the side of the complement automaton, a ranking is an infinite sequence of level rankings in the states of the accepting path. While this seems simple enough conceptually, it requires attention to detail and much technical work in the formalization. The second challenge consists of proving that two ways of stating the same property are equivalent. The last condition in the definition of the odd ranking states that all paths eventually get stuck in an odd rank. On the side of the complement automaton, this property takes the form of a set that keeps track of which paths have yet to visit an odd rank. The acceptance condition of the complement automaton then requires this set to infinitely often become empty, ensuring that no path visits even ranks indefinitely. This again requires coinduction and the construction of infinite paths.

Together with the theorem from the previous section, we obtain the correctness theorem of complementation.

Theorem 1 (Complement Language).

theorem complement_language:
  assumes "finite (nodes A)"
  shows "$\mathcal{L}\ \overline{A} = \Sigma^\omega \setminus \mathcal{L}\ A$"

3.7 Refinement Framework

We want our complementation algorithm and equivalence checker to be executable. When developing formally verified algorithms, there is a trade-off between efficiency of the algorithm and simplicity of the proof. For complex algorithms, a direct proof of an efficient implementation tends to get unmanageable, as implementation details obfuscate the main ideas of the proof.

A standard approach to this problem is stepwise refinement [2], which modularizes the correctness proof. One starts with an abstract version of the algorithm and then refines it in correctness-preserving steps to the concrete, efficient version. A refinement step may reduce the nondeterminism of a program, replace abstract mathematical specifications by concrete algorithms, and replace abstract datatypes by their implementations. For example, selection of an arbitrary element from a set may be refined to getting the head of a list. This approach separates the correctness proof of the algorithm from the correctness proof of the implementation. The former can focus on algorithmic ideas without implementation details getting in the way. The latter consists of a series of refinement steps, each focusing on a specific implementation detail, without having to worry about overall correctness.

In Isabelle/Hol, stepwise refinement is supported by the Refinement Framework [24, 32, 26, 25] and the Isabelle Collections Framework [23, 29]. The former implements a refinement calculus [2] based on a nondeterminism monad [44],

while the latter provides a library of verified efficient data structures. Both frameworks come with tool support to simplify their usage for algorithm development and to automate canonical tasks such as verification condition generation.

3.8 Implementation

Now that the abstract correctness of our complementation procedure is proven, we want to derive an executable algorithm from our definitions. We use the aforementioned refinement framework to refine our definitions that involve partial functions and sets to executable code working on association lists. For instance, the abstract correctness proof is most naturally stated on the complement state type $(Q \rightharpoonup \mathbb{N}) \times Q$ set. However, the isomorphic type $Q \rightharpoonup (\mathbb{N} \times \text{bool})$ is more suitable for the implementation. Thus, this and several other preliminary steps are taken to bring the definition into the correct shape. We also introduce the language-preserving optimization mentioned in section 2.3 at this stage. The correctness proof of this optimization involves establishing a simulation relation between the original automaton and its optimized version.

Once these manual refinement steps are completed, we then use the automatic refinement tool [21, 22]. It allows us to automatically refine an abstract definition to an executable implementation. It does this by instantiating abstract data structures like sets and partial functions with concrete ones like lists, hash sets, and association lists. Since refinement is compositional and the structure of the algorithm is not affected by these substitutions, refinement proofs only have to be done once for each concrete data structure. As many of these data structures have already been formalized in the library, very little has to be proven manually by the user. For instance, choosing to implement a set with a hash set instead of a list can be as simple as adding a type annotation. In particular, none of the refinement proofs have to be adjusted or redone.

At this stage, we have an executable definition that takes a successor function and gives the successor function of the complement automaton. However, we also want to be able to generate the complement automaton as a whole in an explicit representation. To do this, we make use of the Dfs Framework [31, 30]. It comes with a sample instantiation that collects all unique nodes in a graph. We define the graph induced by a given automaton and generate an executable definition of its successor function. We can then run the previously verified Dfs algorithm on this graph to explore the complement automaton. The correctness proof of this algorithm then states that these are indeed all of the reachable states.

The complement automaton now has the state type $(Q \times (\mathbb{N} \times \text{bool}))$ list. This is the association list implementation of the type $Q \rightharpoonup \mathbb{N} \times \text{bool}$ mentioned earlier. Since this type is rather unwieldy, we use the result of the exploration phase to rename all the complement states using natural numbers. We then use the states explored by the Dfs algorithm to collect all of the transitions in the automaton. The end result is an explicit representation of the complement automaton with label type α and state type $\mathbb{N}$. We have $\overline{A} = (\Sigma, I, \delta, F)$ with $\Sigma :: \alpha$ list, $I :: \mathbb{N}$ list, $\delta :: (\mathbb{N} \times \alpha \times \mathbb{N})$ list, and $F :: \mathbb{N}$ list.

3.9 Equivalence

We now want to use our complementation algorithm to build an equivalence checker as outlined in section 2.4. In order to decide language containment and thus equivalence, we still need a product operation and an emptiness check. To this end, we add more operations to the automata library [7]. We already added several operations for deterministic Büchi automata, deterministic co-Büchi automata, and deterministic Rabin automata as part of [39, 9]. We now also add intersection, union, and degeneralization constructions for nondeterministic Büchi automata. Thanks to the new intermediate abstraction layer mentioned in section 3.3, these operations generalize to all other nondeterministic automata in the library. The main challenge here was finding an abstraction for degeneralization that enables sharing this part of the formalization between both deterministic and nondeterministic automata. In the end, this was achieved by stating the main idea of degeneralization on streams rather than automata.

As mentioned in section 2.4, we use an emptiness check based on Gabow's algorithm for strongly-connected components. For this, we reuse a formalization originally developed as part of the Cava model checker [14, 13]. This formalization [28, 27] includes both the abstract correctness proof of the algorithm, as well as executable code. Furthermore, it supports checking emptiness of generalized Büchi automata directly, enabling us to skip the degeneralization step that would usually be necessary after the product. This turns out to be significantly faster.

We now assemble these parts into an equivalence checker and then refine it to be executable. In contrast to complementation, this algorithm is much more compositional, simplifying both the abstract correctness proof and the refinement steps. We ran into one issue with the correctness theorem in the formalization of Gabow's algorithm [28, 27] not being strong enough due to some technicalities. We would like to thank the author Peter Lammich for quickly generalizing the theorem after this issue was discovered.

3.10 Integration

With all the pieces in place, it is now time to integrate everything into a command-line tool. Having refined all of our definitions to be executable, we can already export Sml code from Isabelle. In order for these algorithms to function as part of a stand-alone tool, we need the ability to input and output automata. For this, we have decided to use the Hanoi Omega-Automata format [1], also called Hoa. It is used by other automata tools such as Spot [11], Owl [19], and Goal [42]. The handling of command-line parameters as well as Hoa parsing and printing are implemented manually in Sml. This piece of code wraps the verified algorithm in a command-line tool and is the only unverified part of the final executable.

The result is a command-line tool with two modes of operation: complementation and equivalence checking. Complementation takes an input automaton in the Hoa format and outputs the complement automaton either as a transition list or in the Hoa format. Equivalence checking takes two input automata in the Hoa format and outputs a truth value indicating their equivalence.

Our formalization is available as part of the Archive of Formal Proofs [5].

4 Evaluation

We evaluate the performance of both our complementation implementation and our equivalence checker. As a benchmark for raw complementation performance, we run our implementation on randomly-generated automata. The results are shown in figure 1.

States	Samples	Completion Rate	Average Time
5	393 438	100.00 %	0.006 s
10	41 496	99.98 %	0.110 s
15	15 616	98.30 %	3.112 s
20	16 950	36.58 %	22.695 s

Fig. 1: Complementation Performance. We use Spot's `randaut` tool to generate random automata with a given number of states. We then run our complementation implementation on them. The time limit was set to 60 seconds.

Furthermore, we compare the performance of our complementation implementation to Spot [11] and GOAL [42]. The results are shown in figure 2.

Tool	Completion Rate	Average Time	States
Spot (`--complement --ba`)	100.00 %	0.006 s	12.76
GOAL (`rank -tr`)	84.13 %	0.837 s	91.3
GOAL (`rank -rd`)	69.01 %	5.661 s	6 010
Our Tool	79.36 %	0.683 s	6 010

Fig. 2: Complementation Performance Comparison. We use Spot's `randltl` and `ltl2tgba` tools to generate automata from random LTL formulae. Automata with a state count other than 10 are discarded and the rest is complemented with various tools. Out of 5741 samples, 3962 could be complemented by all tools within the time limit of 60 seconds. To ensure comparability, we use the latter set of automata for the average time and complement states statistics.

Our tool implements the same algorithm as GOAL with the rank decrement option (`rank -rd`), which is also reflected in the identical number of states of the complement automata. However, our implementation has significantly shorter execution times thanks to extensive profiling efforts and use of efficient data structures from the Isabelle Collections Framework [23, 29]. In fact, this effect is so large that it somewhat makes up for the worse asymptotical state complexity when compared to GOAL with the tight rank option (`rank -tr`). The latter has significant startup overhead, but performs better on automata that are difficult to complement. While the performance of Spot is superior to either of the other tools, we want to emphasize that absolute competitiveness with unverified tools is not the goal of our work. As long as our tool is fast enough to process practical examples, it can serve its purpose as a verified reference implementation.

We also evaluate the performance of the equivalence checker. To do so, we generate random LTL formulae and translate them to Büchi automata via both Spot [11] and Owl [19]. We then use our equivalence checker on these automata. The results are shown in figure 3.

States	**Samples**	**Completion Rate**	**Average Time**
$(0, 5]$	73 001	100.00 %	0.004 s
$(5, 10]$	16 024	98.49 %	0.632 s
$(10, 15]$	4 128	88.32 %	3.607 s
$(15, 20]$	1 347	64.88 %	5.203 s
$(20, \infty)$	1 370	39.12 %	8.543 s
total	95 870	97.88 %	0.347 s

Fig. 3: Equivalence Checker Performance. We use Spot's `randltl` tool to generate random LTL formulae. We then use Spot's `ltl2tgba` tool as well as Owl's `ltl2dra` translation in conjunction with Spot's `autfilt` tool to obtain two translations of the same formula. Finally, we use our equivalence checker to check if both automata do indeed have the same language. The time limit was set to 60 seconds. The state count shown is that of the larger of the two automata.

When running the equivalence checker on automata that are not equivalent, the performance is often better. This is due to the fact that the algorithm searches for an accepting cycle in either $A \times \overline{B}$ or $\overline{A} \times B$. As soon as such a cycle is found, it can abort and return a negative answer. Since both complement and product are represented implicitly, this avoids constructing the full state space.

Finally, we use the same testing procedure on translations of the well-known "Dwyer"-patterns [12]. We were able to successfully check 52 out of the 55 formulae with the following exceptions. One formula resulted in automata of sizes 13 and 8, respectively, whose equivalence could not be verified within the time limit of 600 seconds. Two more formulae were successfully translated by Owl's `ltl2dra` translation procedure, but Spot's `autfilt` tool could not translate them to a nondeterministic Büchi automaton within the time limit of 600 seconds. Note that Spot's `autfilt` tool was also set up to simplify the resulting automata, as otherwise, they would quickly grow to be too large. Out of the 52 checked formulae, 49 could be processed in a matter of milliseconds, with two taking about a second and one taking 129 seconds.

From these tests we conclude that the performance of our tool is good enough to serve as a verified reference tool for examples of practical relevance. Note that tools like Spot include many more optimizations and heuristics that enable them to complement into much smaller automata as well as check the equivalence of much larger automata. However, it is not our goal to compete with Spot, but rather to provide a verified reference tool that is fast enough to be useful for testing other tools.

It turns out that we do not have to look far to find an illustration for this point. While gathering data for this section, our equivalence checker discovered a

language mismatch between Spot's and Owl's translation of the same LTL formula. The developers of Owl confirmed that this was indeed a bug in the implementation of its `ltl2dra` translation procedure and promptly fixed it. Manifestation of this issue was very rare, first occurring after about 50 000 randomly-generated formulae. This demonstrates the need for verified reference implementations, as even extensively tested software can still contain undetected issues.

5 Conclusion

We developed a formally verified and executable complementation procedure and equivalence checker. The formal theory acts as a very detailed and machine-checkable description of rank-based complementation. Additionally, our formalization includes executable reference tools. These come with a strong correctness guarantee as everything from the abstract correctness down to the executable SML code is covered by the verification. This high confidence in their correctness justifies their use to test other, unverified tools.

We also contributed additional functionality as well as an improved architecture to the automata library. This emphasizes the software engineering aspect of formal theory development where theories can be reused and become more and more useful as they mature.

For future work, it would be desirable to formalize an algorithm that generates a complement automaton with fewer states. As mentioned in section 2.3, this concerns both asymptotical state complexity as well as performance in practice. It would also be of interest to verify the bounds on asymptotical state complexity.

References

[1] Tomás Babiak et al. "The Hanoi Omega-Automata Format". In: *CAV 2015*. 2015. DOI: `10.1007/978-3-319-21690-4_31`.

[2] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus - A Systematic Introduction*. 1998. DOI: `10.1007/978-1-4612-1674-2`.

[3] Julian Biendarra et al. "Foundational (Co)datatypes and (Co)recursion for Higher-Order Logic". In: *FroCoS 2017*. 2017. DOI: `10.1007/978-3-319-66167-4_1`.

[4] Jasmin Blanchette et al. "Truly Modular (Co)datatypes for Isabelle/HOL". In: *ITP 2014*. 2014. DOI: `10.1007/978-3-319-08970-6_7`.

[5] Julian Brunner. "Büchi Complementation". In: *Archive of Formal Proofs* (2017). URL: `https://www.isa-afp.org/entries/Buchi_Complementation.html`.

[6] Julian Brunner. "Partial Order Reduction". In: *Archive of Formal Proofs* (2018). URL: `https://www.isa-afp.org/entries/Partial_Order_Reduction.html`.

[7] Julian Brunner. "Transition Systems and Automata". In: *Archive of Formal Proofs* (2017). URL: `https://www.isa-afp.org/entries/Transition_Systems_and_Automata.html`.

[8] Julian Brunner and Peter Lammich. "Formal Verification of an Executable LTL Model Checker with Partial Order Reduction". In: *J. Autom. Reasoning* 1 (2018). DOI: 10.1007/s10817-017-9418-4.

[9] Julian Brunner, Benedikt Seidl, and Salomon Sickert. "A Verified and Compositional Translation of LTL to Deterministic Rabin Automata". In: *ITP 2019*. 2019. DOI: 10.4230/LIPIcs.ITP.2019.11.

[10] J. Richard Büchi. "On a decision method in restricted second order arithmetic". In: *Proceedings of the International Congress on Logic, Methodology, and Philosophy of Science, 1960, Berkeley, California, USA*. 1962.

[11] Alexandre Duret-Lutz et al. "Spot 2.0 - A Framework for LTL and ω-Automata Manipulation". In: *ATVA 2016*. 2016. DOI: 10.1007/978-3-319-46520-3_8.

[12] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. "Property specification patterns for finite-state verification". In: *Proceedings of the Second Workshop on Formal Methods in Software Practice, 1998*. 1998. DOI: 10.1145/298595.298598.

[13] Javier Esparza et al. "A Fully Verified Executable LTL Model Checker". In: *CAV 2013*. 2013. DOI: 10.1007/978-3-642-39799-8_31.

[14] Javier Esparza et al. "A Fully Verified Executable LTL Model Checker". In: *Archive of Formal Proofs* (2014). URL: https://www.isa-afp.org/entries/CAVA_LTL_Modelchecker.shtml.

[15] Ehud Friedgut, Orna Kupferman, and Moshe Y. Vardi. "Büchi Complementation Made Tighter". In: *Int. J. Found. Comput. Sci.* 4 (2006). DOI: 10.1142/S0129054106004145.

[16] Harold N. Gabow. "Path-based depth-first search for strong and biconnected components". In: *Inf. Process. Lett.* 3-4 (2000). DOI: 10.1016/S0020-0190(00)00051-X.

[17] Florian Haftmann and Tobias Nipkow. "Code Generation via Higher-Order Rewrite Systems". In: *FLOPS 2010*. 2010. DOI: 10.1007/978-3-642-12251-4_9.

[18] Gerard J. Holzmann, Doron A. Peled, and Mihalis Yannakakis. "On nested depth first search". In: *The Spin Verification System, Proceedings of a DIMACS Workshop, 1996*. 1996. DOI: 10.1090/dimacs/032/03.

[19] Jan Kretínský, Tobias Meggendorfer, and Salomon Sickert. "Owl: A Library for ω-Words, Automata, and LTL". In: *ATVA 2018*. 2018. DOI: 10.1007/978-3-030-01090-4_34.

[20] Orna Kupferman and Moshe Y. Vardi. "Weak alternating automata are not that weak". In: *ACM Trans. Comput. Log.* 3 (2001). DOI: 10.1145/377978.377993.

[21] Peter Lammich. "Automatic Data Refinement". In: *Archive of Formal Proofs* (2013). URL: https://www.isa-afp.org/entries/Automatic_Refinement.shtml.

[22] Peter Lammich. "Automatic Data Refinement". In: *ITP 2013*. 2013. DOI: 10.1007/978-3-642-39634-2_9.

[23] Peter Lammich. "Collections Framework". In: *Archive of Formal Proofs* (2009). URL: https://www.isa-afp.org/entries/Collections.shtml.

[24] Peter Lammich. "Refinement for Monadic Programs". In: *Archive of Formal Proofs* (2012). URL: https://www.isa-afp.org/entries/Refine_Monadic.shtml.

[25] Peter Lammich. "Refinement to Imperative/HOL". In: *ITP 2015*. 2015. DOI: 10.1007/978-3-319-22102-1_17.

[26] Peter Lammich. "The Imperative Refinement Framework". In: *Archive of Formal Proofs* (2016). URL: https://www.isa-afp.org/entries/Refine_Imperative_HOL.shtml.

[27] Peter Lammich. "Verified Efficient Implementation of Gabow's Strongly Connected Component Algorithm". In: *ITP 2014*. 2014. DOI: 10.1007/978-3-319-08970-6_21.

[28] Peter Lammich. "Verified Efficient Implementation of Gabow's Strongly Connected Components Algorithm". In: *Archive of Formal Proofs* (2014). URL: https://www.isa-afp.org/entries/Gabow_SCC.shtml.

[29] Peter Lammich and Andreas Lochbihler. "The Isabelle Collections Framework". In: *ITP 2010*. 2010. DOI: 10.1007/978-3-642-14052-5_24.

[30] Peter Lammich and René Neumann. "A Framework for Verifying Depth-First Search Algorithms". In: *CPP 2015*. 2015. DOI: 10.1145/2676724.2693165.

[31] Peter Lammich and René Neumann. "A Framework for Verifying Depth-First Search Algorithms". In: *Archive of Formal Proofs* (2016). URL: https://www.isa-afp.org/entries/DFS_Framework.shtml.

[32] Peter Lammich and Thomas Tuerk. "Applying Data Refinement for Monadic Programs to Hopcroft's Algorithm". In: *ITP 2012*. 2012. DOI: 10.1007/978-3-642-32347-8_12.

[33] Stephan Merz. "Weak Alternating Automata in Isabelle/HOL". In: *TPHOLs 2000*. 2000. DOI: 10.1007/3-540-44659-1_26.

[34] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. 2002. ISBN: 3-540-43376-7. DOI: 10.1007/3-540-45949-9.

[35] Robert Sachtleben. "Formalisation of an Adaptive State Counting Algorithm". In: *Archive of Formal Proofs* (2019). URL: https://www.isa-afp.org/entries/Adaptive_State_Counting.html.

[36] Robert Sachtleben et al. "A Mechanised Proof of an Adaptive State Counting Algorithm". In: *ICTSS 2019*. 2019. DOI: 10.1007/978-3-030-31280-0_11.

[37] Shmuel Safra. "On the Complexity of omega-Automata". In: *29th Annual Symposium on Foundations of Computer Science, 1988*. 1988. DOI: 10.1109/SFCS.1988.21948.

[38] Sven Schewe. "Büchi Complementation Made Tight". In: *STACS 2009*. 2009. DOI: 10.4230/LIPIcs.STACS.2009.1854.

[39] Benedikt Seidl and Salomon Sickert. "A Compositional and Unified Translation of LTL into ω-Automata". In: *Archive of Formal Proofs* (2019). URL: https://www.isa-afp.org/entries/LTL_Master_Theorem.html.

[40] Stephen F. Siegel. "What's Wrong with On-the-Fly Partial Order Reduction". In: *CAV 2019*. 2019. DOI: 10.1007/978-3-030-25543-5_27.

[41] Ming-Hsien Tsai et al. "State of Büchi Complementation". In: *Logical Methods in Computer Science* 4 (2014). DOI: 10.2168/LMCS-10(4:13)2014.

[42] Yih-Kuen Tsay et al. "GOAL: A Graphical Tool for Manipulating Büchi Automata and Temporal Formulae". In: *TACAS 2007*. 2007. DOI: 10.1007/978-3-540-71209-1_35.

[43] Moshe Y. Vardi. "The Büchi Complementation Saga". In: *STACS 2007*. 2007. DOI: 10.1007/978-3-540-70918-3_2.

[44] Philip Wadler. "Comprehending Monads". In: *Math. Struct. Comput. Sci.* 4 (1992). DOI: 10.1017/S0960129500001560.

Part III

Extra Publications

Appendix C

LTL Translation

This appendix includes a full copy of the following publication.

Julian Brunner, Benedikt Seidl, and Salomon Sickert. “A Verified and Compositional Translation of LTL to Deterministic Rabin Automata”. In: *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*. ed. by John Harrison, John O’Leary, and Andrew Tolmach. Vol. 141. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 11:1–11:19. DOI: 10.4230/LIPIcs.ITP.2019.11

This article is not relevant to the evaluation of this publication-based dissertation. Instead, it is included as the canonical publication of the library for transition systems and automata introduced in chapter 4. It contains the original introduction of the library, including a more detailed motivation and review of existing formalizations.

Copyright

The article in this appendix was published in the open access series LIPIcs. It was published under the Creative Commons Attribution 3.0 Unported (CC BY 3.0) license. It can thus be freely redistributed, including the reproduction as part of this dissertation.

A Verified and Compositional Translation of LTL to Deterministic Rabin Automata

Julian Brunner
Technische Universität München, Germany
julian.brunner@tum.de

Benedikt Seidl
Technische Universität München, Germany
benedikt.seidl@tum.de

Salomon Sickert[1]
Technische Universität München, Germany
salomon.sickert@tum.de

Abstract

We present a formalisation of the unified translation approach from linear temporal logic (LTL) to ω-automata from [19]. This approach decomposes LTL formulas into "simple" languages and allows a clear separation of concerns: first, we formalise the purely logical result yielding this decomposition; second, we develop a generic, executable, and expressive automata library providing necessary operations on automata to re-combine the "simple" languages; third, we instantiate this generic theory to obtain a construction for deterministic Rabin automata (DRA). We extract from this particular instantiation an executable tool translating LTL to DRAs. To the best of our knowledge this is the first verified translation of LTL to DRAs that is proven to be double-exponential in the worst case which asymptotically matches the known lower bound.

2012 ACM Subject Classification Theory of computation → Automata over infinite objects; Theory of computation → Modal and temporal logics; Theory of computation → Interactive proof systems

Keywords and phrases Automata Theory, Automata over Infinite Words, Deterministic Automata, Linear Temporal Logic, Model Checking, Verified Algorithms

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.11

Supplement Material The described Isabelle/HOL development is archived in the "Archive of Formal Proofs" and is split into the entries [10] and [39].

Funding This work was partially funded and supported by the German Research Foundation (DFG) project "Verified Model Checkers" (317422601).

Acknowledgements The authors want to thank Manuel Eberl, Javier Esparza, Lars Hupel, Peter Lammich, and Tobias Nipkow for their helpful comments and technical expertise.

1 Introduction

As time has shown again and again, bugs in hardware and software can have dramatic costs, ranging from monetary damages over destroyed property to life-threatening situations. In order to prevent the introduction of unwanted behaviour into software or hardware designs, an immense amount of testing and debugging is applied. However, for critical systems such methods are not enough, since they simply cannot guarantee the absence of bugs in general. Formal methods offer here a way forward by applying mathematical rigour to detect and rule out unwanted behaviour. Model checking [14] is one of the most successful techniques

[1] Corresponding author

© Julian Brunner, Benedikt Seidl, and Salomon Sickert;
licensed under Creative Commons License CC-BY

10th International Conference on Interactive Theorem Proving (ITP 2019).
Editors: John Harrison, John O'Leary, and Andrew Tolmach; Article No. 11; pp. 11:1–11:19

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

in the area of formal methods. A key component for model checking reactive systems, i.e., non-terminating systems interacting with an open environment, against a temporal specification language, in our case linear temporal logic (LTL), is the translation to a suitable automaton model over infinite words.

Throughout the last decades, a wide variety of translation strategies to different types of ω-automata have been proposed and implemented, e.g. [23, 22, 2, 4, 43, 17]. However, as mentioned before, software development seems to be inherently error-prone, not to mention there might be mistakes in the definition of these constructions themselves. So, how can we trust these implementations to produce the correct automata for identifying bugs or proving their absence? Who watches the watchers?

Exactly that train of thought lead to the development of the CAVA LTL model checker [20] which is verified in Isabelle and exported as an executable tool. The model checker includes a translation from LTL to nondeterministic Büchi automata due to [23]. However, for model checking other structures, such as probabilistic systems, other types of automata are necessary, such as limit-deterministic [44, 15] or deterministic automata [5]. Consequently, there is the need to formalise new translations from scratch which seems wasteful and cumbersome.

It would be desirable to have a separation of concerns: a theory that captures the common essence of LTL for all desired translations and that leaves a small gap to deal with the specifics of a chosen automaton model. The logical framework of [19] sketches an approach to such a modularisation: a theorem decomposing an LTL formula φ into "simple" languages, named $L^1_{\varphi,X}$, $L^2_{X,Y}$, and $L^3_{X,Y}$, such that:

$$\mathcal{L}(\varphi) = \bigcup_{\substack{X \subseteq \nu(\varphi) \\ Y \subseteq \mu(\varphi)}} (L^1_{\varphi,X} \cap L^2_{X,Y} \cap L^3_{X,Y})$$

where X and Y are sets of least- and greatest-fixed operators – hence the names ν and μ – that are subformulas of φ. We will later see a formal definition of these sets. This decomposition outlines a simple strategy to obtain a translation from LTL to our chosen automaton model: first, we define constructions for the "simple" languages; second, we implement two Boolean operations, namely union and intersection, in the automaton model; third, we combine the automata for $L^1_{\varphi,X}$, $L^2_{X,Y}$, and $L^3_{X,Y}$ using these Boolean operations.

Contribution

We provide a formalisation of [19] in Isabelle and contribute the following components: (1) a generic and expressive automata library[2] providing the necessary Boolean operations, (2) a formalisation of the Master Theorem [19] decomposing LTL formulas, (3) a combination of these two components to obtain an executable and verified translation from LTL to deterministic Rabin automata (DRA) of asymptotic optimal size, and (4) an implementation extracted from the Isabelle theory combined with an LTL parser, a verified LTL simplifier, and a serialisation to the HOA format [3], a textual format for ω-automata. Note that the resulting implementation is just one use-case and using the same framework we can also obtain a construction for other types of ω-automata, e.g. nondeterministic Büchi automata (NBA) or deterministic generalised Rabin automata. However, this would exceed the scope and space of this paper.

[2] The scope of the library is actually wider than just the support of ω-automata: automata on finite words and abstract transition systems can also be expressed.

Isabelle/HOL [36] is a proof assistant based on Higher-Order Logic (HOL), which can be thought of as a combination of functional programming and logic. Formalisations done in Isabelle are trustworthy for two reasons: First, Isabelle's LCF architecture guarantees that all proofs are checked using a very small logical core which is rarely modified but tested extensively over time. This reduces the trusted code base to a minimum. Second, bugs in the core rarely lead to accidentally proving false propositions. Bugs that have large effects are easily caught, while the limited applicability of bugs with small effects is unlikely to coincide with a logical mistake in the large-scale structure of the proof. In order to export executable code, we use the Isabelle code generator in conjunction with the monadic refinement framework [26] and automatic refinement [27]. Finally, we use several entries from the "Archive of Formal Proofs" (AFP), a collection of formalisations for Isabelle that are maintained and continuously machine-checked.

Related Work

A substantial amount of work has already been invested into verifying translations from linear temporal logic (LTL) to nondeterministic Büchi automata (NBA[3]): We already mentioned [20] which includes a translation to NBAs following the tableau construction from [23]. Further, the translation proposed by [22], which translates LTL via very-weak alternating automata to NBAs, has been formalised by [25] in HOL4. This work also includes an executable refinement of the abstract algorithm.

Alternating automata have been previously studied in [34] with an application to the translation of LTL to alternating ω-automata. However, the translation from alternating automaton to NBAs is not included. At the other end of the spectrum the publication [17], with the formal proof development archived in [40], presents a direct, verified, and executable translation from LTL to deterministic generalised Rabin automata. However, this construction is only shown to be triple-exponential and thus one exponential larger than the known, optimal lower bound. It is also important to mention that with the help of the Isabelle formalisation errors in the original publication [18] were uncovered and removed for the journal version [17]. This highlights again how important such a rigorous development for verification tools is.

Another interesting point is that the DRA constructions we provide for the "simple" languages can be seen as a version of Brzozowski's derivatives [13] applied to LTL formulas. Derivative-based constructions seem to be more natural in the functional programming paradigm as the work on regular expression equivalence from [37] shows.

Outline

After a brief introduction of the preliminaries in Section 2 we discuss the used automata formalisation in Section 3. We then give an overview of the LTL decomposition results in Section 4 and finally derive an executable LTL to DRA translation in Section 5.

[3] In the context of this paper we do not distinguish minor variations of acceptance conditions and the term NBA includes also nondeterministic generalised Büchi automata as well as transition-based Büchi automata. Similar we use the term DRA also for deterministic generalised Rabin automata.

2 Preliminaries

Locales

Isabelle provides a mechanism for parameterized theory contexts in the form of locales [6]. In a simplified sense, this means that a named context can be defined that is both parameterized by types and terms as well as augmented with assumptions. It is then possible to add various definitions and theorems within this context. Finally, by instantiating the parameters and proving the assumptions, these definitions and theorems also become instantiated and available to the enclosing context.

ω-Words

Let Σ be a finite alphabet. An ω-word w over Σ is an infinite sequence of letters $a_0 a_1 a_2 \ldots$ with $a_i \in \Sigma$ for all $i \geq 0$ and an ω-language is a set of ω-words. We use two different representations for ω-words over a type α: as a function "$\alpha\,\mathsf{word} = \mathsf{nat} \Rightarrow \alpha$" and as a codatatype "$\alpha\,\mathsf{stream} = \alpha \mathbin{\#\#} \alpha\,\mathsf{stream}$". The reason for this division is historic and is due to the fact that the material building on the *LTL* entry [41] predates the development of the codatatype package [7]. Observe that these two types are isomorphic.

The function $\mathsf{prefix}\ i\ w$ returns the finite prefix of w of length i and the function $\mathsf{suffix}\ i\ w$ gives the infinite suffix of w starting at i. The concatenation operator $w' \frown w$ prepends the finite word w' to w.

We introduce the constants scan and sscan for lists and streams, respectively. They work like the identically named function in Haskell, in that they perform a fold with accumulation. That is, they fold over a list or stream and collect the state of the fold at each step and return this collection as a list or stream, respectively. Thus, unlike fold, it is also possible to define this function on infinite sequences.

We also introduce the constant "$\mathsf{infs} :: (\alpha \Rightarrow \mathsf{bool}) \Rightarrow \alpha\ \mathsf{stream} \Rightarrow \mathsf{bool}$" that indicates if a predicate is fulfilled infinitely often in a stream. We will use infs to define acceptance conditions for ω-automata.

Linear Temporal Logic

We base our contribution on the *LTL* entry found in the AFP [41] and extend it where necessary. The datatype we use for LTL syntactically enforces formulas to be in negation normal form. In order to preserve the expressiveness of LTL with negation, we need to include for the $\mathbf{U}$ (Until) operator its dual $\mathbf{R}$ (Release). For the logical decomposition result it is also essential to include $\mathbf{W}$ (Weak-Until) and $\mathbf{M}$ (Strong-Release). As usual we use $\mathbf{F}\ \varphi$ (Eventually) as an abbreviation for $\mathbf{tt}\ \mathbf{U}\ \varphi$ and $\mathbf{G}\ \psi$ (Always) for $\mathbf{ff}\ \mathbf{R}\ \psi$.

▶ **Definition 1** (Linear Temporal Logic).

$$\begin{aligned}\textbf{datatype}\ \ \alpha\,\mathsf{ltl} = \mathbf{tt} \mid \mathbf{ff} \mid \alpha \mid \neg\alpha \mid (\alpha\,\mathsf{ltl}) \wedge (\alpha\,\mathsf{ltl}) \mid (\alpha\,\mathsf{ltl}) \vee (\alpha\,\mathsf{ltl}) \mid \mathbf{X}\ (\alpha\,\mathsf{ltl})\\ \mid (\alpha\,\mathsf{ltl})\ \mathbf{U}\ (\alpha\,\mathsf{ltl}) \mid (\alpha\,\mathsf{ltl})\ \mathbf{R}\ (\alpha\,\mathsf{ltl}) \mid (\alpha\,\mathsf{ltl})\ \mathbf{W}\ (\alpha\,\mathsf{ltl}) \mid (\alpha\,\mathsf{ltl})\ \mathbf{M}\ (\alpha\,\mathsf{ltl})\end{aligned}$$

The type variable α determines the type of the atomic propositions. We write $\mathsf{atoms}\ \varphi$ to refer to the set of atomic propositions in a formula φ. The function $\mathsf{sf}\ \varphi$ computes all subformulas of φ, i.e., all subtrees of its syntax tree. Additionally, we define $\mathsf{subformulas}_\mu\ \varphi$ as set of subformulas of the form $\psi\ \mathbf{U}\ \chi$ or $\psi\ \mathbf{M}\ \chi$, and $\mathsf{subformulas}_\nu\ \varphi$ as the set of subformulas of the form $\psi\ \mathbf{R}\ \chi$ or $\psi\ \mathbf{W}\ \chi$.

▶ **Definition 2** (Semantics). *The entailment relation* $\models \, :: \alpha\,\mathsf{set\,word} \Rightarrow \alpha\,\mathsf{ltl} \Rightarrow \mathsf{bool}$ *is defined recursively as follows:*

$$
\begin{array}{llll}
w \models \mathbf{tt} & & w \models \mathbf{X}\ \varphi & = \mathsf{suffix}\ 1\ w \models \varphi \\
w \not\models \mathbf{ff} & & & \\
w \models a & = a \in w\ 0 & w \models \varphi\ \mathbf{U}\ \psi & = \exists i.\ \mathsf{suffix}\ i\ w \models \psi \wedge (\forall j < i.\ \mathsf{suffix}\ j\ w \models \varphi) \\
w \models \neg a & = a \notin w\ 0 & w \models \varphi\ \mathbf{R}\ \psi & = \forall i.\ \mathsf{suffix}\ i\ w \models \psi \vee (\exists j < i.\ \mathsf{suffix}\ j\ w \models \varphi) \\
w \models \varphi \wedge \psi & = w \models \varphi \wedge w \models \psi & w \models \varphi\ \mathbf{W}\ \psi & = \forall i.\ \mathsf{suffix}\ i\ w \models \varphi \vee (\exists j \leq i.\ \mathsf{suffix}\ j\ w \models \psi) \\
w \models \varphi \vee \psi & = w \models \varphi \vee w \models \psi & w \models \varphi\ \mathbf{M}\ \psi & = \exists i.\ \mathsf{suffix}\ i\ w \models \varphi \wedge (\forall j \leq i.\ \mathsf{suffix}\ j\ w \models \psi)
\end{array}
$$

We define the set of all words over an alphabet Σ *satisfying a formula* φ*:*

$$\mathsf{language}\ \Sigma\ \varphi = \{w.\, w \models \varphi \wedge \mathsf{range}\ w \subseteq \Sigma\}\,.$$

Equivalence Relations over LTL

We define three equivalence relations over LTL formulas: The largest equivalence relation is *language equivalence*. Two formulas are *(language-)equivalent* if they are satisfied by exactly the same words.

A smaller relation is defined by *propositional equivalence*. We interpret an LTL formula φ in propositional logic by treating every subformula that is a literal (a, $\neg a$) or a modal operator ($\mathbf{X}$, $\mathbf{U}$, $\mathbf{M}$, $\mathbf{R}$, $\mathbf{W}$) as a propositional variable. If a set of these subformulas $\mathcal{I}$ is a propositional model for φ, we write $\mathcal{I} \models_p \varphi$. Two formulas are *propositionally equivalent* if are satisfied by the same propositional models.

▶ **Definition 3** (Propositional Semantics). *The propositional entailment relation* $\models_p :: \alpha\,\mathsf{ltl\,set} \Rightarrow \alpha\,\mathsf{ltl} \Rightarrow \mathsf{bool}$ *is defined recursively as follows:*

$$
\begin{array}{llll}
\mathcal{I} \models_p \mathbf{tt} & & \mathcal{I} \models_p \mathbf{X}\ \varphi & = (\mathbf{X}\ \varphi) \in \mathcal{I} \\
\mathcal{I} \not\models_p \mathbf{ff} & & & \\
\mathcal{I} \models_p a & = a \in \mathcal{I} & \mathcal{I} \models_p \varphi\ \mathbf{U}\ \psi & = (\varphi\ \mathbf{U}\ \psi) \in \mathcal{I} \\
\mathcal{I} \models_p \neg a & = (\neg a) \in \mathcal{I} & \mathcal{I} \models_p \varphi\ \mathbf{R}\ \psi & = (\varphi\ \mathbf{R}\ \psi) \in \mathcal{I} \\
\mathcal{I} \models_p \varphi \wedge \psi & = \mathcal{I} \models_p \varphi \wedge \mathcal{I} \models_p \psi & \mathcal{I} \models_p \varphi\ \mathbf{W}\ \psi & = (\varphi\ \mathbf{W}\ \psi) \in \mathcal{I} \\
\mathcal{I} \models_p \varphi \vee \psi & = \mathcal{I} \models_p \varphi \vee \mathcal{I} \models_p \psi & \mathcal{I} \models_p \varphi\ \mathbf{M}\ \psi & = (\varphi\ \mathbf{M}\ \psi) \in \mathcal{I}
\end{array}
$$

Finally, *constants equivalence* is the smallest of the three equivalence relations. We use the function $\mathsf{eval} :: \alpha\,\mathsf{ltl} \Rightarrow \mathsf{tvl}$ with the three-valued logic "$\mathsf{tvl} = \mathsf{Yes} \mid \mathsf{No} \mid \mathsf{Maybe}$". It returns Yes iff φ is propositionally equivalent to $\mathbf{tt}$, and No iff φ is propositionally equivalent to $\mathbf{ff}$, respectively. Otherwise, Maybe is returned. The actual Isabelle formalisation does not refer to propositional equivalence, but in order to simplify the presentation we use the presented characterisation. Two formulas φ and ψ are *constants-equivalent* iff they are (syntactically) identical or "$\mathsf{eval}\ \varphi = \mathsf{eval}\ \psi \neq \mathsf{Maybe}$".

▶ **Definition 4** (Equivalence Relations). *For* $\varphi :: \alpha\,\mathsf{ltl}$ *and* $\psi :: \alpha\,\mathsf{ltl}$*, we define:*

$$
\begin{array}{lcl}
\varphi \sim_l \psi & = & \forall w.\ w \models \varphi \longleftrightarrow w \models \psi \\
\varphi \sim_p \psi & = & \forall \mathcal{I}.\ \mathcal{I} \models_p \varphi \longleftrightarrow \mathcal{I} \models_p \psi \\
\varphi \sim_c \psi & = & (\varphi = \psi \vee (\mathsf{eval}\ \varphi = \mathsf{eval}\ \psi \wedge \mathsf{eval}\ \psi \neq \mathsf{Maybe}))
\end{array}
$$

▶ **Lemma 5** (Order of Equivalence Relations).

$$\sim_c \,\leq\, \sim_p \,\leq\, \sim_l$$

Note that this order also corresponds to the computational complexity, with $\sim_c$ being the easiest to compute and $\sim_l$ the hardest.

3 Transition Systems and Automata

Automata are a popular subject in their own right in theoretical computer science and also have many applications, like regular expression matching and model checking. As such, it suggests itself to formalise these concepts separately and generically as a library to be shared. We first establish our goals for such a library. The deceptively simple term automaton covers a diverse range of objects that need to be supported. These differ in various ways, including but not limited to: successors (deterministic, nondeterministic), labelling (state-labeled, transition-labeled), and acceptance condition (finite, Büchi, Rabin, etc.). For each automaton type, we want to formalise fundamental concepts like path, reachability, and language. We would also like to formalise constructions like Boolean operations (union, intersection, complementation), and degeneralisation of Büchi acceptance conditions. As an overall goal, we want to share as much of the formalisation as possible by keeping it abstract. This avoids duplication and often makes definitions and proofs simpler and more elegant. Finally, we want to do all of this while providing good usability and automation, especially concerning the basic concepts that constitute the foundation of the library.

With these goals in mind, we look at the formalisations that are already available in the Isabelle ecosystem. First off, there are many ad-hoc formalisations of transition systems and automata done as part of other formalisations [40, 21, 1, 31]. Furthermore, there have been a few major formalisations as part of the CAVA project [21], although not all of them were preserved or published. Stephan Merz and Alexander Schimpf formalised NBAs and NGBAs in preliminary work of the CAVA project [38] and then later as part of CAVA itself. Peter Lammich is the author of the current CAVA automata library [28], which includes state-labeled NBAs and NGBAs. These formalisations cover a very specific set of automata, making them convenient to use, but only if one happens to need exactly that type of automaton. Another unpublished formalisation by Thomas Tuerk is more generic and covers DFAs, NFAs, NBAs, and NGBAs. It achieves this genericity by modelling some of these automata as special cases of others, which allows for sharing of definitions and proofs. For instance, a deterministic transition system would naturally be modelled using the type "$\alpha \Rightarrow \rho \Rightarrow \rho$". Alternatively, it can also be treated as a special case of a nondeterministic transition system with the type "$(\rho \times \alpha \times \rho)\,\mathsf{set}$". However, this causes several issues. Firstly, since the type is too weak, a uniqueness predicate on the term level is needed to only allow those transition relations that act like functions. These predicates then have to be carried around in all proofs explicitly, rather than being encoded in the type. Secondly, due to the type being a poor fit, we can no longer do things like folding over the successor function. Lastly, the user is restricted to a single representation, rather than, for instance, being able to choose between explicit ("$(\rho \times \alpha \times \rho)\,\mathsf{set}$") and implicit ("$\alpha \Rightarrow \rho \Rightarrow \rho\,\mathsf{set}$") representations.

We use these experiences to design a new architecture in order to achieve the goals we set earlier. Since our primary goal is sharing via abstraction, this is what will mainly motivate our decisions. There are two observations to be made. Firstly, acceptance conditions are far too diverse and specific to be treated abstractly. Thus, our abstract representation will cover transition systems instead of automata, with acceptance conditions being added on a more concrete level at a later stage. This idea is not new and was in fact used in most of the earlier formalisations as well. Secondly, as mentioned in the previous paragraph, specialisation as a mechanism of abstraction has various issues. Instead, we choose to use the mechanism of instantiation via locales (Section 2), the advantages of which will become apparent in the following sections. Thus, the library formalises *abstract transition systems* (Section 3.1), which are then instantiated and used as building blocks for *concrete automata* (Section 3.2).

We try to formalise as much as possible in the context of abstract transition systems, since this both often leads to elegance and conciseness and is shared between all concrete automata. Thanks to this, adding a new automaton requires only a minimal amount of setup, allowing users to use the library in conjunction with their own custom automata representations. That being said, the set of automata supplied with the library is also growing and becoming more useful, making this less and less necessary. In the end, we supply both a collection of useful automata as well as the tools to easily add custom ones as needed.

3.1 Abstract Transition Systems

Having decided on our architecture, the central decision lies in the specification of the locale for transition systems. We focus on the defining property of a transition system: its ability to use transitions to move from state to state. This leads us directly to the specification in terms of its types (*transition* and *state*) and its terms (execute and enabled).

▶ **Definition 6.**

locale transition-system =

fixes execute :: *transition* ⇒ *state* ⇒ *state*

fixes enabled :: *transition* ⇒ *state* ⇒ bool

Given a transition and a source state, the function execute specifies the target state for that transition. Analogously, the function enabled determines whether the given transition is enabled at the given source state. Together, these functions capture the essence of a transition system in terms of its ability to transition between states. Given the types, it may seem appealing to combine both constants into a single one with result type "*state* option". This sounds great in theory, but unfortunately, is very inconvenient to work with in practice. It mixes the issue of finding the target of a transition with that of whether that transition was valid in the first place. Keeping these two things separate makes definitions simpler and allows for better automation in proofs.

Having defined the transition-system locale, we now develop some abstract theory within this context. So far, we can only execute single transitions, so we look at finite and infinite sequences of transitions. We introduce the following constants based on the execute function.

▶ **Definition 7.**

target = fold execute :: *transition* list ⇒ *state* ⇒ *state*

trace = scan execute :: *transition* list ⇒ *state* ⇒ *state* list

strace = sscan execute :: *transition* stream ⇒ *state* ⇒ *state* stream

Given a sequence of transitions and a source state, these functions give the target state and the finite and infinite sequence of traversed states, respectively. Note both the simplicity and elegance of these definitions and how each of them is simply a lifted version of execute.

We can do something similar for the enabled function.

▶ **Definition 8.**

inductive path :: *transition* list ⇒ *state* ⇒ bool **where**

path [] p

enabled $a\ p \implies$ path r (execute $a\ p$) $\implies$ path $(a \# r)\ p$

coinductive spath :: *transition* stream ⇒ *state* ⇒ bool **where**

enabled $a\ p \implies$ spath r (execute $a\ p$) $\implies$ spath $(a \#\# r)\ p$

These constants are (co)inductively defined predicates that capture the notion of all the transitions in a sequence being enabled at their respective states. Like in the previous paragraph, these are basically lifted versions of enabled, which is also reflected in their types.

Together, these form the very foundation of the library, since almost every other concept is in some way related to sequences of transitions. The nice thing about these definitions is that they lend themselves very well to automation. In the case of definitions lifted from execute, we can define simplification rules. In the case of definitions lifted from enabled, we can define safe introduction and elimination rules. This works for both the constructors of sequences (# and ##), as well as the operators for concatenation (@, @−). Convenience and automation regarding the basic concepts was a major shortcoming of earlier libraries.

Next, we define the constant reachable for the set of reachable states from a source state. Like path, this is an inductively defined predicate. Alternatively, we could have defined reachable in terms of target and path. Instead, it is defined directly based on execute and enabled and the connection to target and path is shown as a lemma.

There are some interesting things we can formalise even on this very abstract level. We present one such example in the construction of infinite paths.

▶ **Lemma 9** (Recurring Condition).

$$\begin{aligned}
\textbf{fixes} \quad & P :: \mathit{state} \Rightarrow \mathsf{bool} \;\; \textbf{and} \;\; p :: \mathit{state} \\
\textbf{assumes} \quad & P\ p \;\; \textbf{and} \;\; \bigwedge p.\ P\ p \implies \exists r.\ r \neq [] \wedge \mathsf{path}\ r\ p \wedge P\ (\mathsf{target}\ r\ p) \\
\textbf{obtains} \quad & r :: \mathit{transition}\,\mathsf{stream} \\
\textbf{where} \quad & \mathsf{spath}\ r\ p \;\; \textbf{and} \;\; \mathsf{infs}\ P\ (p\ \#\#\ \mathsf{strace}\ r\ p)
\end{aligned}$$

Here, the premises only guarantee the repeated existence of a finite extension to an existing finite path, which we want to use to construct an infinite path. Proving a statement like this is cumbersome, as it requires skolemisation of the premise, construction of a stream via iteration combinators and finally proving the properties via coinduction. By providing generic rules like these, all this complexity is hidden and users can restrict themselves to easy-to-work-with constants like spath and infs.

3.2 Concrete Automata

In order for our formalisation of abstract transition systems to be useful, it needs to be able to express a wide range of transition system types and their representations. We now present instantiations for various transition systems with labels of type α and states of type ρ.

Given a successor function "$\mathsf{succ} :: \alpha \Rightarrow \rho \Rightarrow \rho\,\mathsf{option}$", we instantiate as follows.

▶ **Example 10** (Incomplete Deterministic Transition System).

$$\mathsf{execute} = \lambda a\ p.\ \mathsf{the}\ (\mathsf{succ}\ a\ p) \qquad \mathit{transition} = \alpha$$
$$\mathsf{enabled} = \lambda a\ p.\ \mathsf{succ}\ a\ p \neq \mathsf{None} \qquad \mathit{state} = \rho$$

Note how the deterministic successor function fits the interface straightforwardly.

Given a successor function "$\mathsf{succ} :: \alpha \Rightarrow \rho \Rightarrow \rho$" we instantiate as follows.

▶ **Example 11** (Complete Deterministic Transition System).

$$\mathsf{execute} = \mathsf{succ} \qquad \mathit{transition} = \alpha$$
$$\mathsf{enabled} = \top \qquad \mathit{state} = \rho$$

Things get interesting when considering "$\mathsf{succ} :: \alpha \Rightarrow \rho \Rightarrow \rho\,\mathsf{set}$". Textbooks teach us that deterministic transitions systems are a special case of nondeterministic ones. At first glance, it may seem like we are trying to do the impossible opposite here. However, since we get to instantiate the type variables, there is a surprisingly elegant solution.

▶ **Example 12** (Implicit Nondeterministic Transition System)**.**

$$\mathsf{execute} = \lambda(a, q)\ p.\ q \qquad\qquad \mathit{transition} = \alpha \times \rho$$
$$\mathsf{enabled} = \lambda(a, q)\ p.\ q \in \mathsf{succ}\ a\ p \qquad\qquad \mathit{state} = \rho$$

Note how unlike in the first two examples, the type variable *transition* gets instantiated in a nontrivial way. While it may seem backwards at first, this actually works out perfectly and gives our constants the strongest possible type for this scenario. For instance, we get "$\mathsf{path} :: (\alpha \times \rho)\,\mathsf{list} \Rightarrow \rho \Rightarrow \mathsf{bool}$". That is, the path predicate expects a source state as well as a list of the traversed labels and states. This expression contains exactly the necessary amount of information, nothing more, nothing less. Note that the fact that we are dealing with pairs is not an issue, as Isabelle has good automation for those. We also added some more automation for sequences of pairs as part of this library. In the end, neither the deterministic nor the nondeterministic case necessitates inconvenient wellformedness predicates while sharing the same abstract formalisation.

Finally, we consider an explicit representation in "$\mathsf{trans} :: (\rho \times \alpha \times \rho)\,\mathsf{set}$". Being isomorphic to the previous case, the type variables as well as execute are instantiated the same way.

▶ **Example 13** (Explicit Nondeterministic Transition System)**.**

$$\mathsf{execute} = \lambda(a, q)\ p.\ q \qquad\qquad \mathit{transition} = \alpha \times \rho$$
$$\mathsf{enabled} = \lambda(a, q)\ p.\ (p, a, q) \in \mathsf{trans} \qquad\qquad \mathit{state} = \rho$$

Unsurprisingly, an isomorphic change in representation does not make a difference since the instantiation absorbs such details.

Having shown that we can instantiate a variety of transition systems using our abstract theory, we can now use these as building blocks for concrete automata. Since the abstraction is achieved via type instantiation and locales, it only minimally impacts the usability compared to a fully specific formalisation. Moreover, since it does not restrict the type of the automaton at all, the user can use a representation that exactly fits their needs.

There are some definitions that one would expect to be part of a general automata library that unfortunately cannot be formalised on transition systems. One of these are Boolean operations, since they require information about the automaton's successors, labelling, and acceptance condition. With some effort, they could be formalised on intermediate abstraction over a family of similar automata (for instance, DBA, DCA, DRA). However, we could not justify the effort for our purposes, since these formalisations do not contain much substance.

Degeneralisation, which plays an important part in defining aforementioned Boolean operations, can be generalised a little easier. The reason for this is that it is independent from successors and labelling, requiring only the concept of state-based Büchi acceptance. Thanks to this, we were able to abstractly formalise degeneralisation in a transition system locale augmented with an acceptance condition. This intermediate abstraction is then instantiated in order to facilitate the formalisation of Boolean operations on DBAs and DCAs.

3.3 Predefined Automata

While the focus of the automata library is on the abstract part and the provision of tools to build concrete automata, it also comes with a growing collection of the latter. At the time of writing, it contains (non)deterministic finite automata, (non)deterministic Büchi automata, as well as deterministic co-Büchi and Rabin automata. Each of these incurs around 50 lines of proof text in order to set-up the automaton and to define its language. The latter is fairly simple to achieve, as all the constituents (paths and acceptance conditions) are already available and just need to be composed to yield a language definition.

3.4 Executable Implementation

One of our goals is also the ability to implement executable versions of some algorithms. As mentioned earlier, we will use the refinement frameworks and the Isabelle code generator for this. Most of this needs to be done on concrete automata, as it depends on details of the representation. Furthermore, in many cases it is advantageous to be able to choose data structures depending on the representation. Because of these reasons, all the executable implementations are done on the concrete level, with only some proofs being reused.

We build on existing algorithms for graph structures to implement versions that work with automata. For instance, we use the AFP entry about depth-first search [32, 33] to explore all reachable states of an automaton. This is used to generate explicit representations of automata in order to be able to serialise and output them. In the case of NBAs we consider the successor function "$\mathsf{succ} :: \alpha \Rightarrow \rho \Rightarrow \rho\, \mathsf{set}$", which implicitly represents the transitions of the automaton. The algorithm can then turn this into an explicit set of transitions "$\mathsf{trans} :: (\rho \times \alpha \times \rho)\, \mathsf{set}$". We also implement an algorithm for translating an automaton with an arbitrary state type into one whose states are natural numbers. Furthermore, we use the AFP entry about Gabow's algorithm for strongly-connected components [29, 30] to decide language emptiness of NBAs.

3.5 Formalisation

The library is available in the form of the AFP entry *Transition Systems and Automata* [10]. At the time of writing, it comprises about 5800 lines of theory text. Other than in this paper, the library is used in the partial order reduction optimisation [12, 11] of the CAVA model checker [21]. It is also used as the foundation of the AFP entry about rank-based complementation of Büchi automata [9].

3.6 Contributions to the Translation Formalisation

For this paper, we contribute deterministic Büchi, co-Büchi, and Rabin automata. For instance, the constructor for deterministic Büchi automata is "$\mathsf{dba} :: \alpha\, \mathsf{set} \Rightarrow \rho \Rightarrow (\alpha \Rightarrow \rho \Rightarrow \rho) \Rightarrow (\rho \Rightarrow \mathsf{bool}) \Rightarrow (\alpha, \rho)\ \mathsf{dba}$". Furthermore, we add corresponding union and intersection operations to the library (Figure 1). In addition to those operations, we also implement a specialised operation dbcrai that provides the intersection of a DBA and a DCA resulting in a DRA. We prove both their correctness in terms of language as well as upper bounds on the number of states of the resulting automata. Since the resulting automata are implicit, we also provide an executable algorithm for exploration and subsequent conversion to an explicit representation together with a numbering of the states.

Automaton	$\cap$ (Pair)	$\bigcap$ (List)	$\cup$ (Pair)	$\bigcup$ (List)
DBA		dbail	dbau	dbaul
DCA	dcai	dcail		dcaul
DRA				draul

Figure 1 Boolean Operations on Deterministic ω-Automata. Shown are the Boolean operations that were implemented for deterministic Büchi, co-Büchi, and Rabin automata.

4 The Master Theorem: Decomposing LTL Formulas

The centrepiece for all translations is the *Master Theorem* [19] that decomposes LTL formulas into a Boolean combination, in our case union and intersection, of "simple" languages. We will recall important definitions from [19] in order to state the theorem itself and to highlight obstacles we encountered in our formalisation. For an in-depth discussion and exposition of the theory and its proof we refer the reader to the primary source [19].

We will now introduce the functions used in the scope of the Master Theorem: the "after"-function $\mathsf{af}\ \varphi\ w$, read "φ after w", and the two "advice" functions $\varphi[X]_\nu$ and $\psi[Y]_\mu$ which are pronounced as "φ with **GF**-advice X" and "ψ with **FG**-advice Y", respectively.

4.1 The "after"-Function

Let us begin with the definition of the "after"-function [18, 17, 19]. The function application $\mathsf{af}\ \varphi\ w$ computes a new formula such that for every infinite word w' we have:

▶ **Lemma 14** ([19]).

$$w \frown w' \models \varphi \iff w' \models \mathsf{af}\ \varphi\ w.$$

We can intuitively see af as a function that returns a formula representing the language that we obtain *after* reading the prefix w. We achieve this by using well-known LTL expansion rules combined with partial evaluation.

▶ **Definition 15** ("after"-Function [19]). *The function* $\mathsf{af} :: \alpha\,\mathsf{ltl} \Rightarrow \alpha\,\mathsf{set} \Rightarrow \alpha\,\mathsf{ltl}$ *is defined for a single letter recursively as follows:*

$$\begin{aligned}
&\mathsf{af}\ \mathtt{tt}\ \sigma = \mathtt{tt} && \mathsf{af}\ (\mathbf{X}\ \varphi)\ \sigma = \varphi\\
&\mathsf{af}\ \mathtt{ff}\ \sigma = \mathtt{ff} && \\
&\mathsf{af}\ a\ \sigma = \textbf{if}\ a \in \sigma\ \textbf{then}\ \mathtt{tt}\ \textbf{else}\ \mathtt{ff} && \mathsf{af}\ (\varphi\,\mathbf{U}\,\psi)\ \sigma = (\mathsf{af}\ \psi\ \sigma) \vee ((\mathsf{af}\ \varphi\ \sigma) \wedge (\varphi\,\mathbf{U}\,\psi))\\
&\mathsf{af}\ (\neg a)\ \sigma = \textbf{if}\ a \notin \sigma\ \textbf{then}\ \mathtt{tt}\ \textbf{else}\ \mathtt{ff} && \mathsf{af}\ (\varphi\,\mathbf{R}\,\psi)\ \sigma = (\mathsf{af}\ \psi\ \sigma) \wedge ((\mathsf{af}\ \varphi\ \sigma) \vee (\varphi\,\mathbf{R}\,\psi))\\
&\mathsf{af}\ (\varphi \wedge \psi)\ \sigma = (\mathsf{af}\ \varphi\ \sigma) \wedge (\mathsf{af}\ \psi\ \sigma) && \mathsf{af}\ (\varphi\,\mathbf{W}\,\psi)\ \sigma = (\mathsf{af}\ \psi\ \sigma) \vee ((\mathsf{af}\ \varphi\ \sigma) \wedge (\varphi\,\mathbf{W}\,\psi))\\
&\mathsf{af}\ (\varphi \vee \psi)\ \sigma = (\mathsf{af}\ \varphi\ \sigma) \vee (\mathsf{af}\ \psi\ \sigma) && \mathsf{af}\ (\varphi\,\mathbf{M}\,\psi)\ \sigma = (\mathsf{af}\ \psi\ \sigma) \wedge ((\mathsf{af}\ \varphi\ \sigma) \vee (\varphi\,\mathbf{M}\,\psi))
\end{aligned}$$

We generalise this definition to finite words by overloading $\mathsf{af} :: \alpha\,\mathsf{ltl} \Rightarrow \alpha\,\mathsf{set}\,\mathsf{list} \Rightarrow \alpha\,\mathsf{ltl}$*:*

$$\mathsf{af}\ \varphi\ w = \mathsf{foldl}\ \mathsf{af}\ \varphi\ w.$$

▶ Remark 16. The reader might have noticed that the definition of af resembles the idea of Brzozowski's derivatives for regular expressions [13]. In fact, as we will see later, the DRA construction relies on af and the previously introduced LTL equivalence relations again mirroring the idea of Brzozowski. However, this approach alone can only be applied to fragments of LTL.

4.2 Syntactic Fragments of LTL

We already teased the idea of the "simple" languages, but what is special about these? What is the mechanism to achieve this? These languages are made simple by the fact that they can be expressed by fragments of LTL. To be more precise, let μLTL be the fragment that only contains modal operators that can be expressed as least-fixed points, i.e., we disallow the operators $\mathbf{R}$ and $\mathbf{W}$. Dually, νLTL contains only modal operators that can be expressed as greatest-fixed points, i.e., we disallow the operators $\mathbf{U}$ and $\mathbf{M}$. The fragments $\mathbf{GF}(\mu\text{LTL})$ and $\mathbf{FG}(\nu\text{LTL})$ contain all formulas $\mathbf{GF}\varphi$ and $\mathbf{FG}\psi$ where $\varphi \in \mu$LTL and $\psi \in \nu$LTL, respectively. For these fragments one can easily define translations to NBAs or DRAs, e.g. [19].

Let us now think about how to make use of this: Assume one gets a promise set $X = \{a \,\mathbf{U}\, b\}$ guaranteeing that $a \,\mathbf{U}\, b$ holds infinitely often, i.e., $w \models \mathbf{GF}(a \,\mathbf{U}\, b)$, and assume we have access to a translation for νLTL. Can we simplify $\varphi = \mathbf{G}(a \,\mathbf{U}\, b) \vee \mathbf{G}c$ with this information? Since $w \models \mathbf{GF}(a \,\mathbf{U}\, b)$ implies that b is infinitely often true, we can replace the $\mathbf{U}$ by an $\mathbf{W}$. Under the assumption that X is a correct promise, we simplify φ to an equivalent formula $\mathbf{G}(a \,\mathbf{W}\, b) \vee \mathbf{G}c$ which is a formula of νLTL. Then we can apply our translation for the νLTL fragment.

Formally, we define the functions $\varphi[X]_\nu$ and $\varphi[Y]_\mu$ such that $\varphi[X]_\nu$ takes a promise set X and produces a formula of νLTL, and such that $\varphi[Y]_\mu$ takes a promise set Y and produces a formula of μLTL:

▶ **Definition 17** ("Advice"-Functions [19]). *The function* $\cdot[\,\cdot\,]_\nu :: \alpha\,\mathsf{ltl} \Rightarrow \alpha\,\mathsf{ltl\,set} \Rightarrow \alpha\,\mathsf{ltl}$ *is defined for the cases* $\mathbf{U}$ *and* $\mathbf{M}$ *as follows:*

$$(\varphi \,\mathbf{U}\, \psi)[X]_\nu = \textbf{if } (\varphi \,\mathbf{U}\, \psi) \in X \textbf{ then } (\varphi[X]_\nu) \,\mathbf{W}\, (\psi[X]_\nu) \textbf{ else ff}$$
$$(\varphi \,\mathbf{M}\, \psi)[X]_\nu = \textbf{if } (\varphi \,\mathbf{M}\, \psi) \in X \textbf{ then } (\varphi[X]_\nu) \,\mathbf{R}\, (\psi[X]_\nu) \textbf{ else ff}$$

The function $\cdot[\,\cdot\,]_\mu :: \alpha\,\mathsf{ltl} \Rightarrow \alpha\,\mathsf{ltl\,set} \Rightarrow \alpha\,\mathsf{ltl}$ *is defined for the cases* $\mathbf{R}$ *and* $\mathbf{W}$ *as follows:*

$$(\varphi \,\mathbf{R}\, \psi)[Y]_\mu = \textbf{if } (\varphi \,\mathbf{R}\, \psi) \in Y \textbf{ then tt else } (\varphi[Y]_\mu) \,\mathbf{M}\, (\psi[Y]_\mu)$$
$$(\varphi \,\mathbf{W}\, \psi)[Y]_\mu = \textbf{if } (\varphi \,\mathbf{W}\, \psi) \in Y \textbf{ then tt else } (\varphi[Y]_\mu) \,\mathbf{U}\, (\psi[Y]_\mu)$$

For all other cases, both functions are defined as a recursive descent over the syntax tree.

4.3 The Master Theorem

We are now equipped with the necessary definitions to state the Master Theorem. Note that the formulation we use is taken nearly verbatim from the Isabelle theory, apart from the annotations $L^1_{\varphi,X}$, $L^2_{X,Y}$, and $L^3_{X,Y}$ that we added to relate to the introduction.

▶ **Theorem 18** (Master Theorem [19]).

$$\begin{aligned} w \models \varphi \iff (\exists X \subseteq \mathsf{subformulas}_\mu\ \varphi.\ \exists Y \subseteq \mathsf{subformulas}_\nu\ \varphi.& \\ (\exists i.\ \mathsf{suffix}\ i\ w \models \mathsf{af}\ \varphi\ (\mathsf{prefix}\ i\ w)[X]_\nu) & \quad — L^1_{\varphi,X} \\ \wedge (\forall \psi \in X.\ w \models \mathbf{G}\ (\mathbf{F}\ \psi[Y]_\mu)) & \quad — L^2_{X,Y} \\ \wedge (\forall \psi \in Y.\ w \models \mathbf{F}\ (\mathbf{G}\ \psi[X]_\nu)) & \quad — L^3_{X,Y} \end{aligned}$$

The proof of this theorem intrinsically depends on the fact that we can check promise sets bottom-up, as formalised by the following lemma. We highlight this intermediate lemma, because we needed to introduce a custom induction mechanism over finite sets to our theory. The remaining material needed to show Theorem 18 is obtained in straight-forward manner and closely resembles the proofs of [19].

▶ **Lemma 19** ([19]).

$$\begin{aligned}
\textbf{fixes}\ & w :: \alpha\,\mathsf{set\,word}\ \ \textbf{and}\ \ \varphi :: \alpha\,\mathsf{ltl}\\
\textbf{assumes}\ & X \subseteq \mathsf{subformulas}_\mu\ \varphi\ \ \textbf{and}\ \ Y \subseteq \mathsf{subformulas}_\nu\ \varphi\\
\textbf{and}\ & \forall\psi \in X.\ w \models \mathbf{G}\ (\mathbf{F}\ \psi[Y]_\mu)\ \ \textbf{and}\ \ \forall\psi \in Y.\ w \models \mathbf{F}\ (\mathbf{G}\ \psi[X]_\nu)\\
\textbf{shows}\ & \forall\psi \in X.\ w \models \mathbf{G}\ (\mathbf{F}\ \psi)\ \ \textbf{and}\ \ \forall\psi \in Y.\ w \models \mathbf{F}\ (\mathbf{G}\ \psi)
\end{aligned}$$

The corresponding proof from [19] proceeds by constructing a sequence of pairs (X_i, Y_i) where we have $(X_0, Y_0) = (\emptyset, \emptyset)$ and $(X_n, Y_n) = (X, Y)$. Moreover, in each step a single formula $\psi_i \in X \uplus Y$ is added to either X_i or Y_i, depending on whether $\psi_i \in X$ or $\psi_i \in Y$. However, ψ_i cannot be chosen arbitrarily and ψ_i must respect the subformula order, i.e., if $\psi_i \in \mathsf{sf}\ \psi_j$, then $i \leq j$. Then the proof proceeds by an induction over this sequence.

Since to the best of our knowledge there has been at the time of writing no matching induction rule in Isabelle or its libraries, we derived a suitable induction rule for our purposes. First, note that instead of sorting the formulas by the subformula order, it is sufficient to order them by their size, because all subformulas of a formula φ are smaller than φ. Second, an induction over pairs of sets seemed inconvenient to us in the context of our theorem prover. Hence we combined the two disjoint sets into a single one and used a suitable case distinction. Finally, we arrived at the following, general induction rule[4] for finite sets with an additional order constraint:

▶ **Lemma 20** (Finite Ordered Induction).

$$\begin{aligned}
\textbf{fixes}\ & S :: \alpha\,\mathsf{set}\ \ \textbf{and}\ \ P :: \alpha\,\mathsf{set} \Rightarrow \mathsf{bool}\ \ \textbf{and}\ \ f :: \alpha \Rightarrow (\beta :: \mathsf{linorder})\\
\textbf{assumes}\ & \mathsf{finite}\ S\ \ \textbf{and}\ \ P\ \emptyset\\
\textbf{and}\ & \bigwedge x\ S.\ \mathsf{finite}\ S \wedge (\forall y.\ y \in S \longrightarrow f\ y \leq f\ x) \wedge P\ S \implies P\ (\mathsf{insert}\ x\ S)\\
\textbf{shows}\ & P\ S
\end{aligned}$$

5 Deriving the DRA Construction

With the necessary decomposition theorem in place, we now can follow our automata construction blue-print to obtain a translation from LTL to DRAs. We will first build automata for $L^1_{\varphi,X}$, $L^2_{X,Y}$, and $L^3_{X,Y}$, named $\mathfrak{A}_1$, $\mathfrak{A}_2$, and $\mathfrak{A}_3$, respectively. In the subsequent section, we will assemble these pieces to the final automaton and end the section with a description of the extracted, verified tool.

5.1 Constructing Automata for $L^1_{\varphi,X}$, $L^2_{X,Y}$, and $L^3_{X,Y}$

We parametrise our automata constructions for the "simple" components by an equivalence relation $\sim$. The most important requirement for $\sim$ is that $\sim_c \leq \sim \leq \sim_l$ holds, i.e., that $\sim$ does not consider two formulas with different languages equivalent and $\sim$ eventually detects equivalence to **tt** and **ff** for certain fragments. This abstraction has two advantages over fixing a concrete equivalence: first, our proofs stay as abstract as possible and the proof automation does not rely accidentally on irrelevant properties of the chosen equivalence relation; second, we can instantiate the final automaton with any suitable equivalence relation. In Section 5.3 we exemplarily use propositional equivalence but one can easily replace it by a different equivalence without any additional effort to speak of.

[4] This induction rule has now been included in Isabelle/HOL, is located in `HOL/Lattices_Big.thy`, and is named `finite_ranking_induct`.

In this paper, we will only discuss the construction of $\mathfrak{A}_2$ for $L^2_{X,Y}$. The constructions for $L^1_{\varphi,X}$ and $L^3_{X,Y}$ as defined by [19] are formalised analogously. Remember that $L^2_{X,Y}$ is defined as "$\bigcap \psi \in X.$ language UNIV $(\mathbf{GF}(\psi[Y]_\mu))$" for the finite sets X and Y. Hence it suffices to define a translation for formulas of the fragment $\mathbf{GF}(\mu\text{LTL})$ and then apply the intersection construction from the automaton library.

For the translation of formulas from the fragment $\mathbf{GF}(\mu\text{LTL})$ we make use of the following lemma. It states the we can monitor a formula from μLTL using af and the constrained equivalence relation $\sim$, and if a word satisfies the formula, then we will notice this after a finite amount of steps. Furthermore, the lemma states that we can deal with $\mathbf{GF}(\mu\text{LTL})$ by repeatedly doing this:

▶ **Lemma 21** (Logical Characterisation of μLTL and $\mathbf{GF}(\mu\text{LTL})$ [19, 42][5]).

$$\begin{aligned} &\textbf{assumes } \varphi \in \mu\text{LTL} \textbf{ and } \sim_c \leq \sim \leq \sim_l \\ &\quad \textbf{shows } w \models \varphi \iff \exists i.\ \mathsf{af}\ \varphi\ (\mathsf{prefix}\ i\ w) \sim \mathtt{tt} \\ &\qquad \textbf{and } w \models \mathbf{G}\ (\mathbf{F}\ \varphi) \iff \forall i.\ \exists j.\ \mathsf{af}\ (\mathbf{F}\ \varphi)\ (\mathsf{prefix}\ j\ (\mathsf{suffix}\ i\ w)) \sim \mathtt{tt} \end{aligned}$$

Since $\sim$ is such a fundamental ingredient throughout the formalisation of the automata constructions, we use locales in Isabelle to fix $\sim$ and assumptions about it. In particular, we use the equivalence classes of $\sim$ as states in our constructed automata. To define the quotient type for a given equivalence relation, we use the Isabelle's *Quotient* package introduced in [8] and revised in [24]. However, it is not possible to define such a quotient type within a locale. Thus we present a primitive, ad-hoc mechanism to simulate the quotient type in our locale. We fix a type parameter γ and the functions Rep and Abs that compute the representative of an equivalence class and the equivalence class of a formula, respectively. In other words we use Rep and Abs to map between equivalence classes and representatives. Further, we assume the quotient type invariant "Abs (Rep x) $= x$" and require that equality on γ is equivalent to $\sim$ on formulas. Thus we can pretend γ to be a quotient type over $\sim$ which resembles "duck typing" found in programming languages such as Python.

▶ **Definition 22** (Locale for LTL to DRA translation[6]).

$$\begin{aligned} &\textbf{locale } \text{ltl-to-dra} = \\ &\qquad\qquad \textbf{fixes } \sim :: \alpha\ \mathsf{ltl} \Rightarrow \alpha\ \mathsf{ltl} \Rightarrow \mathsf{bool} \\ &\qquad\qquad \textbf{and } \mathsf{Rep} :: \gamma \Rightarrow \alpha\ \mathsf{ltl} \textbf{ and } \mathsf{Abs} :: \alpha\ \mathsf{ltl} \Rightarrow \gamma \\ &\qquad \textbf{assumes } \mathsf{equivp} \sim \textbf{ and } \sim_c \leq \sim \leq \sim_l \\ &\qquad\qquad \textbf{and } \mathsf{Abs}\ (\mathsf{Rep}\ x) = x \textbf{ and } \mathsf{Abs}\ \varphi = \mathsf{Abs}\ \psi \longleftrightarrow \varphi \sim \psi \\ &\qquad\qquad \textbf{and } \varphi \sim \psi \implies (\mathsf{af}\ \varphi\ \sigma \sim \mathsf{af}\ \psi\ \sigma) \wedge (\varphi[X]_\nu \sim \psi[X]_\nu) \end{aligned}$$

In this definition two new assumptions can be found that we have not talked about yet: We also demand that af and $\cdot\,[\,\cdot\,]_\nu$ are congruent with respect to $\sim$. This is due to the fact that our the automata use equivalence classes as states and for computing the successor with af the choice of the representative must be irrelevant.

[5] This lemma is a generalised version of [19] which only considers the special case for $\sim_p$.
[6] We only present the final combination of several locales defined in our Isabelle formalisation to give an overview of all assumptions required by our proofs.

Within this locale we now define the deterministic Büchi automaton $\mathfrak{A}_\mu^{\mathbf{GF}}$ for a single formula of the fragment $\mathbf{GF}(\mu\text{LTL})$. The DBA $\mathfrak{A}_2$ for $L^2_{X,Y}$ is then computed by a Büchi intersection (dbail). Note that this intersection construction requires the operands to be ordered. Hence we represent the advice sets X and Y as the lists xs and ys and propagate this order to dbail.

▶ Definition 23.

$$\mathfrak{A}_\mu^{\mathbf{GF}}\ \varphi = \mathsf{dba\ UNIV\ (Abs\ (}\mathbf{F}\ \varphi))\ (\mathsf{af}_F\ \varphi)\ (\lambda\psi.\ \psi = \mathsf{Abs}\ \mathbf{tt})$$
$$\mathsf{af}_F\ \varphi\ \sigma\ \psi = \mathbf{if}\ \ \psi = \mathsf{Abs}\ \mathbf{tt}\ \ \mathbf{then}\ \ \mathsf{Abs}\ (\mathbf{F}\ \varphi)\ \ \mathbf{else}\ \ \mathsf{Abs}\ (\mathsf{af}\ (\mathsf{Rep}\ \psi)\ \sigma)$$
$$\mathfrak{A}_2\ xs\ ys = \mathsf{dbail}\ (\mathsf{map}\ (\lambda\psi.\ \mathfrak{A}_\mu^{\mathbf{GF}}\ (\psi[\mathsf{set}\ ys]_\mu))\ xs)$$

Using Lemma 21 we show correctness for a single component and using the lemmas from the automata library we also prove the intersection correct. The constructions for $L^1_{\varphi,X}$ and $L^3_{X,Y}$ are analogous and thus skipped from the presentation in this paper.

5.2 Assembling the Pieces

It now remains to intersect the (co-)Büchi automata "$\mathfrak{A}_1\ \varphi\ xs$", "$\mathfrak{A}_2\ xs\ ys$", and "$\mathfrak{A}_3\ xs\ ys$", representing $L^1_{\varphi,X}$, $L^2_{X,Y}$, and $L^3_{X,Y}$, respectively. Again we need to use a list representation for X and Y to fix an iteration order and thus we use xs and ys. We call the resulting Rabin automaton "$\mathfrak{A}\ \varphi\ xs\ ys$". To finish the construction, we then iterate over all possible choices for $X \subseteq \mathsf{subformulas}_\mu\ \varphi$ and $Y \subseteq \mathsf{subformulas}_\nu\ \varphi$ and take the union of all languages accepted by "$\mathfrak{A}\ \varphi\ xs\ ys$" with draul (DRA union):

▶ Definition 24.

$$\mathsf{ltl\text{-}to\text{-}dra}\ \varphi = \mathsf{draul}\ (\mathsf{map}\ (\lambda(xs, ys).\ \mathfrak{A}\ \varphi\ xs\ ys)\ (\mathsf{advice\text{-}sets}\ \varphi)).$$

Using the Master Theorem (Theorem 18) and the correctness lemmas for the intermediate constructions, we obtain the correctness of the translation:

▶ Theorem 25.

$$\mathsf{language}\ (\mathsf{ltl\text{-}to\text{-}dra}\ \varphi) = \mathsf{language\ UNIV}\ \varphi.$$

5.3 A Verified LTL Translator

We extract the executable translation of LTL formulas into ω-automata by instantiating the locale with a suitable equivalence relation. As mentioned above we use $\sim_p$ and we show for this equivalence relation that the constructed automaton indeed has at most a double-exponential number of states in the size of the formula. Hence an exploration by depth-first search terminates, and more importantly, this makes the construction the first LTL to DRA translation with a formally verified double exponential size bound.

▶ Lemma 26.

$$\mathsf{card}\ (\mathsf{nodes}\ (\mathsf{ltl\text{-}to\text{-}dra}\ \varphi)) \leq 2\ \hat{}\ 2\ \hat{}\ (2 * \mathsf{size}\ \varphi + \mathsf{floorlog}\ 2\ (\mathsf{size}\ \varphi) + 4).$$

Exporting code for the LTL part needs only minor adjustments through code lemmas, e.g. we instantiate $\sim_p$ with code provided by [35]. For the parts related to automata we rely on the code export feature of the automata library, see Section 3.4. Notice that Theorem 25

refers to the potentially infinite alphabet UNIV. Choosing UNIV as the alphabet simplified the proofs leading up to the result, but potentially infinite alphabets make an exploration using depth-first search using a naive enumeration of letters impossible. Consequently, we restrict the alphabet to a finite set for the code export by only considering atomic propositions occurring in φ. The resulting constant ltl-to-draei has the signature α ltl $\Rightarrow$ (α set, nat) draei which is then exported to Standard ML. The overall correctness theorem is as follows:

▶ **Theorem 27.**

language (draei-dra (ltl-to-draei φ)) = language (Pow (atoms φ)) φ.

Note that the constant language is only defined for DRAs with a transition function (dra) while we obtain from the translation a DRA with a list of transitions (draei). The constant draei-dra converts an automaton of type draei back to one of type dra.

In the final tool, we combine the function ltl-to-draei with an unverified LTL parser and an unverified serialisation to the Hanoi Omega Automata format [3], a text-based format for representing ω-automata. It is then compiled with `mlton` or `polyc` using the build scripts included in the formalisation [39].

▶ **Example 28.** The following command translates the formula $\mathbf{FG}a$ to a DRA in HOA format and then, using `autfilt` from Spot [16], prints it in the `dot`-format. The result gets rendered by `dot` and is written to a PDF file.

```
./ltl_to_dra "F G a" | autfilt --dot --merge-transitions | dot -Tpdf -O
```

6 Concluding Remarks

The formalisation of the "Master Theorem" itself did not pose major obstacles and did not require special care except for the mentioned techniques. However, the LTL entry [41] and dependencies are host to several LTL datatypes and matching lemmas and notation. This excessive amount of copy-pasting is due the inability to define fragments of datatypes, i.e., restrictions on the constructors used. While one could use **typedef** to carve out restricted types using a predicate, this new type misses the structure of the type we started with. Thus we choose in some cases to have separate datatypes connected by translations, while in other cases we used simple predicates to capture fragments. We think the addition of a mechanism addressing this issue – the definition of datatype fragments and the addition of necessary constants and proof automation – would be worthwhile, since we conjecture it would significantly reduce the size and complexity of LTL related theories.

There are several topics we want to investigate going forward: First, we also want to derive constructions for NBAs and LDBAs. Second, we plan to reduce the size of the generated automata by restricting the possible choices for the advice sets X and Y. Third, we want to provide implementations using better instantiations for the equivalence relation to further reduce the size of the computed automata. Fourth, provide constructions for DRA variants, e.g., transition-based or generalised acceptance. Fifth, while adding some of the Boolean operations, we realised that constructions for ω-automata could potentially be shared and consolidated in an intermediate abstraction.

References

1 Romain Aïssat, Frédéric Voisin, and Burkhart Wolff. Infeasible Paths Elimination by Symbolic Execution Techniques: Proof of Correctness and Preservation of Paths. *Archive of Formal Proofs*, 2016, 2016. URL: https://www.isa-afp.org/entries/InfPathElimination.shtml.

2. Tomás Babiak, Thomas Badie, Alexandre Duret-Lutz, Mojmír Křetínský, and Jan Strejcek. Compositional Approach to Suspension and Other Improvements to LTL Translation. In Ezio Bartocci and C. R. Ramakrishnan, editors, *Model Checking Software - 20th International Symposium, SPIN 2013, Stony Brook, NY, USA, July 8-9, 2013. Proceedings*, volume 7976 of *Lecture Notes in Computer Science*, pages 81–98. Springer, 2013. doi:10.1007/978-3-642-39176-7_6.
3. Tomás Babiak, Frantisek Blahoudek, Alexandre Duret-Lutz, Joachim Klein, Jan Křetínský, David Müller, David Parker, and Jan Strejcek. The Hanoi Omega-Automata Format. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 479–486. Springer, 2015. doi:10.1007/978-3-319-21690-4_31.
4. Tomás Babiak, Frantisek Blahoudek, Mojmír Křetínský, and Jan Strejcek. Effective Translation of LTL to Deterministic Rabin Automata: Beyond the (F, G)-Fragment. In Dang Van Hung and Mizuhito Ogawa, editors, *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*, volume 8172 of *Lecture Notes in Computer Science*, pages 24–39. Springer, 2013. doi:10.1007/978-3-319-02444-8_4.
5. Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.
6. Clemens Ballarin. Locales: A Module System for Mathematical Theories. *J. Autom. Reasoning*, 52(2):123–153, 2014. doi:10.1007/s10817-013-9284-7.
7. Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Truly Modular (Co)datatypes for Isabelle/HOL. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8558 of *Lecture Notes in Computer Science*, pages 93–110. Springer, 2014. doi:10.1007/978-3-319-08970-6_7.
8. Maksym Bortin and Christoph Lüth. Structured Formal Development with Quotient Types in Isabelle/HOL. In Serge Autexier, Jacques Calmet, David Delahaye, Patrick D. F. Ion, Laurence Rideau, Renaud Rioboo, and Alan P. Sexton, editors, *Intelligent Computer Mathematics, 10th International Conference, AISC 2010, 17th Symposium, Calculemus 2010, and 9th International Conference, MKM 2010, Paris, France, July 5-10, 2010. Proceedings*, volume 6167 of *Lecture Notes in Computer Science*, pages 34–48. Springer, 2010. doi:10.1007/978-3-642-14128-7_5.
9. Julian Brunner. Büchi complementation. *Archive of Formal Proofs*, 2017, 2017. URL: https://www.isa-afp.org/entries/Buchi_Complementation.html.
10. Julian Brunner. Transition Systems and Automata. *Archive of Formal Proofs*, 2017, 2017. URL: https://www.isa-afp.org/entries/Transition_Systems_and_Automata.html.
11. Julian Brunner. Partial Order Reduction. *Archive of Formal Proofs*, 2018, 2018. URL: https://www.isa-afp.org/entries/Partial_Order_Reduction.html.
12. Julian Brunner and Peter Lammich. Formal Verification of an Executable LTL Model Checker with Partial Order Reduction. *J. Autom. Reasoning*, 60(1):3–21, 2018. doi:10.1007/s10817-017-9418-4.
13. Janusz A. Brzozowski. Derivatives of Regular Expressions. *J. ACM*, 11(4):481–494, 1964. doi:10.1145/321239.321249.
14. Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018. doi:10.1007/978-3-319-10575-8.
15. Costas Courcoubetis and Mihalis Yannakakis. The Complexity of Probabilistic Verification. *J. ACM*, 42(4):857–907, 1995. doi:10.1145/210332.210339.
16. Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0 - A Framework for LTL and ω-Automata Manipulation. In Cyrille Artho, Axel Legay, and Doron Peled, editors, *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*, volume 9938 of *Lecture Notes in Computer Science*, pages 122–129, 2016. doi:10.1007/978-3-319-46520-3_8.

17 Javier Esparza, Jan Křeínský, and Salomon Sickert. From LTL to deterministic automata - A safraless compositional approach. *Formal Methods in System Design*, 49(3):219–271, 2016. doi:10.1007/s10703-016-0259-2.

18 Javier Esparza and Jan Křetínský. From LTL to Deterministic Automata: A Safraless Compositional Approach. In Armin Biere and Roderick Bloem, editors, *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*, volume 8559 of *Lecture Notes in Computer Science*, pages 192–208. Springer, 2014. doi:10.1007/978-3-319-08867-9_13.

19 Javier Esparza, Jan Křetínský, and Salomon Sickert. One Theorem to Rule Them All: A Unified Translation of LTL into ω-Automata. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 384–393. ACM, 2018. doi:10.1145/3209108.3209161.

20 Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. A Fully Verified Executable LTL Model Checker. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 463–478. Springer, 2013. doi:10.1007/978-3-642-39799-8_31.

21 Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. A Fully Verified Executable LTL Model Checker. *Archive of Formal Proofs*, 2014, 2014. URL: https://www.isa-afp.org/entries/CAVA_LTL_Modelchecker.shtml.

22 Paul Gastin and Denis Oddoux. Fast LTL to Büchi Automata Translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2001. doi:10.1007/3-540-44585-4_6.

23 Rob Gerth, Doron A. Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In Piotr Dembinski and Marek Sredniawa, editors, *Protocol Specification, Testing and Verification XV, Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification, Warsaw, Poland, June 1995*, volume 38 of *IFIP Conference Proceedings*, pages 3–18. Chapman & Hall, 1995.

24 Brian Huffman and Ondrej Kuncar. Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL. In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, volume 8307 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2013. doi:10.1007/978-3-319-03545-1_9.

25 Simon Jantsch and Michael Norrish. Verifying the LTL to Büchi Automata Translation via Very Weak Alternating Automata. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10895 of *Lecture Notes in Computer Science*, pages 306–323. Springer, 2018. doi:10.1007/978-3-319-94821-8_18.

26 Peter Lammich. Refinement for Monadic Programs. *Archive of Formal Proofs*, 2012, 2012. URL: https://www.isa-afp.org/entries/Refine_Monadic.shtml.

27 Peter Lammich. Automatic Data Refinement. *Archive of Formal Proofs*, 2013, 2013. URL: https://www.isa-afp.org/entries/Automatic_Refinement.shtml.

28 Peter Lammich. The CAVA Automata Library. *Archive of Formal Proofs*, 2014, 2014. URL: https://www.isa-afp.org/entries/CAVA_Automata.shtml.

29 Peter Lammich. Verified Efficient Implementation of Gabow's Strongly Connected Component Algorithm. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8558 of *Lecture Notes in Computer Science*, pages 325–340. Springer, 2014. doi:10.1007/978-3-319-08970-6_21.

30 Peter Lammich. Verified Efficient Implementation of Gabow's Strongly Connected Components Algorithm. *Archive of Formal Proofs*, 2014, 2014. URL: `https://www.isa-afp.org/entries/Gabow_SCC.shtml`.

31 Peter Lammich and Markus Müller-Olm. Formalization of Conflict Analysis of Programs with Procedures, Thread Creation, and Monitors. *Archive of Formal Proofs*, 2007, 2007. URL: `https://www.isa-afp.org/entries/Program-Conflict-Analysis.shtml`.

32 Peter Lammich and René Neumann. A Framework for Verifying Depth-First Search Algorithms. In Xavier Leroy and Alwen Tiu, editors, *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*, pages 137–146. ACM, 2015. `doi:10.1145/2676724.2693165`.

33 Peter Lammich and René Neumann. A Framework for Verifying Depth-First Search Algorithms. *Archive of Formal Proofs*, 2016, 2016. URL: `https://www.isa-afp.org/entries/DFS_Framework.shtml`.

34 Stephan Merz. Weak Alternating Automata in Isabelle/HOL. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics, 13th International Conference, TPHOLs 2000, Portland, Oregon, USA, August 14-18, 2000, Proceedings*, volume 1869 of *Lecture Notes in Computer Science*, pages 424–441. Springer, 2000. `doi:10.1007/3-540-44659-1_26`.

35 Tobias Nipkow. Boolean Expression Checkers. *Archive of Formal Proofs*, 2014, 2014. URL: `https://www.isa-afp.org/entries/Boolean_Expression_Checkers.shtml`.

36 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. `doi:10.1007/3-540-45949-9`.

37 Tobias Nipkow and Dmitriy Traytel. Unified Decision Procedures for Regular Expression Equivalence. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8558 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2014. `doi:10.1007/978-3-319-08970-6_29`.

38 Alexander Schimpf, Stephan Merz, and Jan-Georg Smaus. Construction of Büchi Automata for LTL Model Checking Verified in Isabelle/HOL. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 424–439. Springer, 2009. `doi:10.1007/978-3-642-03359-9_29`.

39 Benedikt Seidl and Salomon Sickert. A Compositional and Unified Translation of LTL into ω-Automata. *Archive of Formal Proofs*, 2019, 2019. URL: `https://isa-afp.org/entries/LTL_Master_Theorem.html`.

40 Salomon Sickert. Converting Linear Temporal Logic to Deterministic (Generalised) Rabin Automata. *Archive of Formal Proofs*, 2015, 2015. URL: `https://www.isa-afp.org/entries/LTL_to_DRA.shtml`.

41 Salomon Sickert. Linear Temporal Logic. *Archive of Formal Proofs*, 2016, 2016. URL: `https://www.isa-afp.org/entries/LTL.shtml`.

42 Salomon Sickert. *A Unified Translation of Linear Temporal Logic to ω-Automata*. PhD thesis, Technical University Munich, Germany, 2019.

43 Salomon Sickert, Javier Esparza, Stefan Jaax, and Jan Křetínský. Limit-Deterministic Büchi Automata for Linear Temporal Logic. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 312–332. Springer, 2016. `doi:10.1007/978-3-319-41540-6_17`.

44 Moshe Y. Vardi. Automatic Verification of Probabilistic Concurrent Finite-State Programs. In *26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985*, pages 327–338. IEEE Computer Society, 1985. `doi:10.1109/SFCS.1985.12`.

Part IV

Bibliography

[AVW16] Romain Aïssat, Frédéric Voisin, and Burkhart Wolff. "Infeasible Paths Elimination by Symbolic Execution Techniques: Proof of Correctness and Preservation of Paths". In: *Archive of Formal Proofs* 2016 (2016). URL: https://www.isa-afp.org/entries/InfPathElimination.shtml.

[Bab+13a] Tomás Babiak, Thomas Badie, Alexandre Duret-Lutz, Mojmír Kretínský, and Jan Strejcek. "Compositional Approach to Suspension and Other Improvements to LTL Translation". In: *Model Checking Software - 20th International Symposium, SPIN 2013, Stony Brook, NY, USA, July 8-9, 2013. Proceedings*. Ed. by Ezio Bartocci and C. R. Ramakrishnan. Vol. 7976. Lecture Notes in Computer Science. Springer, 2013, pp. 81–98. DOI: 10.1007/978-3-642-39176-7_6.

[Bab+13b] Tomás Babiak, Frantisek Blahoudek, Mojmír Kretínský, and Jan Strejcek. "Effective Translation of LTL to Deterministic Rabin Automata: Beyond the (F, G)-Fragment". In: *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*. Ed. by Dang Van Hung and Mizuhito Ogawa. Vol. 8172. Lecture Notes in Computer Science. Springer, 2013, pp. 24–39. DOI: 10.1007/978-3-319-02444-8_4.

[Bab+15] Tomás Babiak, Frantisek Blahoudek, Alexandre Duret-Lutz, Joachim Klein, Jan Kretínský, 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*. Ed. by Daniel Kroening and Corina S. Pasareanu. Vol. 9206. Lecture Notes in Computer Science. Springer, 2015, pp. 479–486. DOI: 10.1007/978-3-319-21690-4_31.

[Bal14] Clemens Ballarin. "Locales: A Module System for Mathematical Theories". In: *Journal of Automated Reasoning* 52.2 (2014), pp. 123–153. DOI: 10.1007/s10817-013-9284-7.

[Bie+17] Julian Biendarra, Jasmin Christian Blanchette, Aymeric Bouzy, Martin Desharnais, Mathias Fleury, Johannes Hölzl, Ondrej Kuncar, Andreas Lochbihler, Fabian Meier, Lorenz Panny, Andrei Popescu, Christian Sternagel, René Thiemann, and Dmitriy Traytel. "Foundational (Co)datatypes and (Co)recursion for Higher-Order Logic". In: *Frontiers of Combining Systems - 11th International Symposium, FroCoS 2017, Brasília, Brazil, September*

27-29, 2017, Proceedings. Ed. by Clare Dixon and Marcelo Finger. Vol. 10483. Lecture Notes in Computer Science. Springer, 2017, pp. 3–21. DOI: 10.1007/978-3-319-66167-4_1.

[BL16] Julian Brunner and Peter Lammich. "Formal Verification of an Executable LTL Model Checker with Partial Order Reduction". In: *NASA Formal Methods - 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings*. Ed. by Sanjai Rayadurgam and Oksana Tkachuk. Vol. 9690. Lecture Notes in Computer Science. Springer, 2016, pp. 307–321. DOI: 10.1007/978-3-319-40648-0_23.

[BL18] Julian Brunner and Peter Lammich. "Formal Verification of an Executable LTL Model Checker with Partial Order Reduction". In: *Journal of Automated Reasoning* 60.1 (2018), pp. 3–21. DOI: 10.1007/s10817-017-9418-4.

[Bla+14] Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. "Truly Modular (Co)datatypes for Isabelle/HOL". In: *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*. Ed. by Gerwin Klein and Ruben Gamboa. Vol. 8558. Lecture Notes in Computer Science. Springer, 2014, pp. 93–110. DOI: 10.1007/978-3-319-08970-6_7.

[Bru14] Julian Brunner. "Implementation and Verification of Partial Order Reduction for On-The-Fly Model Checking". MA thesis. Technische Universität München, July 15, 2014. 83 pp. URL: http://www21.in.tum.de/~brunnerj/documents/ivporotfmc.pdf.

[Bru17a] Julian Brunner. "Büchi Complementation". In: *Archive of Formal Proofs* 2017 (2017). URL: https://www.isa-afp.org/entries/Buchi_Complementation.html.

[Bru17b] Julian Brunner. "Transition Systems and Automata". In: *Archive of Formal Proofs* 2017 (2017). URL: https://www.isa-afp.org/entries/Transition_Systems_and_Automata.html.

[Bru18] Julian Brunner. "Partial Order Reduction". In: *Archive of Formal Proofs* 2018 (2018). URL: https://www.isa-afp.org/entries/Partial_Order_Reduction.html.

[Bru20] Julian Brunner. "Formal Verification of Executable Complementation and Equivalence Checking for Büchi Automata". In: *Integrated Formal Methods - 16th International Conference, IFM 2020, Lugano, Switzerland, November 16-20, 2020, Proceedings*. Ed. by Brijesh Dongol and Elena Troubitsyna. Vol. 12546. Lecture Notes in Computer Science. Springer, 2020, pp. 239–256. DOI: 10.1007/978-3-030-63461-2_13.

[BSS19] Julian Brunner, Benedikt Seidl, and Salomon Sickert. "A Verified and Compositional Translation of LTL to Deterministic Rabin Automata". In: *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*. Ed. by John Harrison, John O'Leary, and Andrew Tolmach. Vol. 141. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 11:1–11:19. DOI: 10.4230/LIPIcs.ITP.2019.11.

[Büc62] J. Richard Büchi. "On a Decision Method in Restricted Second Order Arithmetic". In: *Proceedings of the International Congress on Logic, Methodology, and Philosophy of Science, 1960, Berkeley, California, USA*. Stanford University Press, 1962, pp. 1–12.

[BW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus - A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998. ISBN: 978-0-387-98417-9. DOI: 10.1007/978-1-4612-1674-2.

[CGP01] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 2001. ISBN: 978-0-262-03270-4. URL: http://books.google.de/books?id=Nmc4wEaLXFEC.

[Cho74] Yaacov Choueka. "Theories of Automata on omega-Tapes: A Simplified Approach". In: *Journal of Computer and System Sciences* 8.2 (1974), pp. 117–141. DOI: 10.1016/S0022-0000(74)80051-6.

[Cla+18] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, eds. *Handbook of Model Checking*. Springer, 2018. ISBN: 978-3-319-10574-1. DOI: 10.1007/978-3-319-10575-8.

[Cou+92] Costas Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. "Memory-Efficient Algorithms for the Verification of Temporal Properties". In: *Formal Methods in System Design* 1.2/3 (1992), pp. 275–288. DOI: 10.1007/BF00121128.

[CP96] Ching-Tsun Chou and Doron A. Peled. “Formal Verification of a Partial-Order Reduction Technique for Model Checking”. In: *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop, TACAS ’96, Passau, Germany, March 27-29, 1996, Proceedings*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 1055. Lecture Notes in Computer Science. Springer, 1996, pp. 241–257. DOI: 10.1007/3-540-61042-1_48.

[DAC98] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. “Property specification patterns for finite-state verification”. In: *Proceedings of the Second Workshop on Formal Methods in Software Practice, March 4-5, 1998, Clearwater Beach, Florida, USA*. Ed. by Mark A. Ardis and Joanne M. Atlee. ACM, 1998, pp. 7–15. DOI: 10.1145/298595.298598.

[Dur+16] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. “Spot 2.0 - A Framework for LTL and ω-Automata Manipulation”. In: *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*. Ed. by Cyrille Artho, Axel Legay, and Doron Peled. Vol. 9938. Lecture Notes in Computer Science. 2016, pp. 122–129. DOI: 10.1007/978-3-319-46520-3_8.

[EKS16] Javier Esparza, Jan Kretínský, and Salomon Sickert. “From LTL to deterministic automata - A safraless compositional approach”. In: *Formal Methods in System Design* 49.3 (2016), pp. 219–271. DOI: 10.1007/s10703-016-0259-2.

[EKS18] Javier Esparza, Jan Kretínský, and Salomon Sickert. “One Theorem to Rule Them All: A Unified Translation of LTL into ω-Automata”. In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*. Ed. by Anuj Dawar and Erich Grädel. ACM, 2018, pp. 384–393. DOI: 10.1145/3209108.3209161.

[Esp+13] Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. “A Fully Verified Executable LTL Model Checker”. In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 463–478. DOI: 10.1007/978-3-642-39799-8_31.

[Esp+14] Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. "A Fully Verified Executable LTL Model Checker". In: *Archive of Formal Proofs* 2014 (2014). URL: https://www.isa-afp.org/entries/CAVA_LTL_Modelchecker.shtml.

[Ete99] Kousha Etessami. "Stutter-Invariant Languages, omega-Automata, and Temporal Logic". In: *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*. Ed. by Nicolas Halbwachs and Doron A. Peled. Vol. 1633. Lecture Notes in Computer Science. Springer, 1999, pp. 236–248. DOI: 10.1007/3-540-48683-6_22.

[FKV06] Ehud Friedgut, Orna Kupferman, and Moshe Y. Vardi. "Büchi Complementation Made Tighter". In: *International Journal of Foundations of Computer Science* 17.4 (2006), pp. 851–868. DOI: 10.1142/S0129054106004145.

[Gab00] Harold N. Gabow. "Path-based depth-first search for strong and biconnected components". In: *Information Processing Letters* 74.3-4 (2000), pp. 107–114. DOI: 10.1016/S0020-0190(00)00051-X.

[Ger+95] Rob Gerth, Doron A. Peled, Moshe Y. Vardi, and Pierre Wolper. "Simple on-the-fly automatic verification of linear temporal logic". In: *Protocol Specification, Testing and Verification XV, Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification, Warsaw, Poland, June 1995*. Ed. by Piotr Dembinski and Marek Sredniawa. Vol. 38. IFIP Conference Proceedings. Chapman & Hall, 1995, pp. 3–18.

[GO01] Paul Gastin and Denis Oddoux. "Fast LTL to Büchi Automata Translation". In: *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*. Ed. by Gérard Berry, Hubert Comon, and Alain Finkel. Vol. 2102. Lecture Notes in Computer Science. Springer, 2001, pp. 53–65. DOI: 10.1007/3-540-44585-4_6.

[God90] Patrice Godefroid. "Using Partial Orders to Improve Automatic Verification Methods". In: *Computer Aided Verification, 2nd International Workshop, CAV '90, New Brunswick, NJ, USA, June 18-21, 1990, Proceedings*. Ed. by Edmund M. Clarke and Robert P. Kurshan. Vol. 531. Lecture Notes in Computer Science. Springer, 1990, pp. 176–185. DOI: 10.1007/BFb0023731.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. Vol. 1032. Lecture Notes in Computer Science. Springer, 1996. ISBN: 3-540-60761-7. DOI: 10.1007/3-540-60761-7.

[HK96] Gerard J. Holzmann and Orna Kupferman. "Not checking for closure under stuttering". In: *The Spin Verification System, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, August, 1996*. Ed. by Jean-Charles Grégoire, Gerard J. Holzmann, and Doron A. Peled. Vol. 32. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. DIMACS/AMS, 1996, pp. 17–22. DOI: 10.1090/dimacs/032/02.

[HN10] Florian Haftmann and Tobias Nipkow. "Code Generation via Higher-Order Rewrite Systems". In: *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings*. Ed. by Matthias Blume, Naoki Kobayashi, and Germán Vidal. Vol. 6009. Lecture Notes in Computer Science. Springer, 2010, pp. 103–117. DOI: 10.1007/978-3-642-12251-4_9.

[Hol04] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004. ISBN: 978-0-321-22862-8.

[Hol97] Gerard J. Holzmann. "The Model Checker SPIN". In: *IEEE Transactions on Software Engineering* 23.5 (1997), pp. 279–295. DOI: 10.1109/32.588521.

[HPY96] Gerard J. Holzmann, Doron A. Peled, and Mihalis Yannakakis. "On nested depth first search". In: *The Spin Verification System, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, August, 1996*. Ed. by Jean-Charles Grégoire, Gerard J. Holzmann, and Doron A. Peled. Vol. 32. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. DIMACS/AMS, 1996, pp. 23–31. DOI: 10.1090/dimacs/032/03.

[Jac06] Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006. ISBN: 978-0-262-10114-1. URL: http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=10928.

[KMS18] Jan Kretínský, Tobias Meggendorfer, and Salomon Sickert. "Owl: A Library for ω-Words, Automata, and LTL". In: *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10,*

2018, Proceedings. Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 11138. Lecture Notes in Computer Science. Springer, 2018, pp. 543–550. DOI: 10.1007/978-3-030-01090-4_34.

[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*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 585–591. DOI: 10.1007/978-3-642-22110-1_47.

[Kur+98] Robert P. Kurshan, Vladimir Levin, Marius Minea, Doron A. Peled, and Hüsnü Yenigün. "Static Partial Order Reduction". In: *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*. Ed. by Bernhard Steffen. Vol. 1384. Lecture Notes in Computer Science. Springer, 1998, pp. 345–357. DOI: 10.1007/BFb0054182.

[Kur94] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes. The Automata-Theoretic Approach*. Princeton Series in Computer Science. Princeton University Press, 1994. ISBN: 978-0691034362.

[KV01] Orna Kupferman and Moshe Y. Vardi. "Weak alternating automata are not that weak". In: *ACM Transactions on Computational Logic* 2.3 (2001), pp. 408–429. DOI: 10.1145/377978.377993.

[Lam09] Peter Lammich. "Collections Framework". In: *Archive of Formal Proofs* 2009 (2009). URL: https://www.isa-afp.org/entries/Collections.shtml.

[Lam12] Peter Lammich. "Refinement for Monadic Programs". In: *Archive of Formal Proofs* 2012 (2012). URL: https://www.isa-afp.org/entries/Refine_Monadic.shtml.

[Lam13a] Peter Lammich. "Automatic Data Refinement". In: *Archive of Formal Proofs* 2013 (2013). URL: https://www.isa-afp.org/entries/Automatic_Refinement.shtml.

[Lam13b] Peter Lammich. "Automatic Data Refinement". In: *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*. Ed. by Sandrine Blazy,

Christine Paulin-Mohring, and David Pichardie. Vol. 7998. Lecture Notes in Computer Science. Springer, 2013, pp. 84–99. DOI: 10.1007/978-3-642-39634-2_9.

[Lam14a] Peter Lammich. “The CAVA Automata Library”. In: *Archive of Formal Proofs* 2014 (2014). URL: https://www.isa-afp.org/entries/CAVA_Automata.shtml.

[Lam14b] Peter Lammich. “Verified Efficient Implementation of Gabow’s Strongly Connected Component Algorithm”. In: *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*. Ed. by Gerwin Klein and Ruben Gamboa. Vol. 8558. Lecture Notes in Computer Science. Springer, 2014, pp. 325–340. DOI: 10.1007/978-3-319-08970-6_21.

[Lam14c] Peter Lammich. “Verified Efficient Implementation of Gabow’s Strongly Connected Components Algorithm”. In: *Archive of Formal Proofs* 2014 (2014). URL: https://www.isa-afp.org/entries/Gabow_SCC.shtml.

[Lam15] Peter Lammich. “Refinement to Imperative/HOL”. In: *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*. Ed. by Christian Urban and Xingyuan Zhang. Vol. 9236. Lecture Notes in Computer Science. Springer, 2015, pp. 253–269. DOI: 10.1007/978-3-319-22102-1_17.

[Lam16] Peter Lammich. “The Imperative Refinement Framework”. In: *Archive of Formal Proofs* 2016 (2016). URL: https://www.isa-afp.org/entries/Refine_Imperative_HOL.shtml.

[LL10] Peter Lammich and Andreas Lochbihler. “The Isabelle Collections Framework”. In: *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*. Ed. by Matt Kaufmann and Lawrence C. Paulson. Vol. 6172. Lecture Notes in Computer Science. Springer, 2010, pp. 339–354. DOI: 10.1007/978-3-642-14052-5_24.

[LM07] Peter Lammich and Markus Müller-Olm. “Formalization of Conflict Analysis of Programs with Procedures, Thread Creation, and Monitors”. In: *Archive of Formal Proofs* 2007 (2007). URL: https://www.isa-afp.org/entries/Program-Conflict-Analysis.shtml.

[LN15] Peter Lammich and René Neumann. “A Framework for Verifying Depth-First Search Algorithms”. In: *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*. Ed. by Xavier Leroy and Alwen Tiu. ACM, 2015, pp. 137–146. DOI: 10.1145/2676724.2693165.

[LN16] Peter Lammich and René Neumann. “A Framework for Verifying Depth-First Search Algorithms”. In: *Archive of Formal Proofs* 2016 (2016). URL: https://www.isa-afp.org/entries/DFS_Framework.shtml.

[Loc10] Andreas Lochbihler. “Coinductive”. In: *Archive of Formal Proofs* 2010 (2010). URL: https://www.isa-afp.org/entries/Coinductive.shtml.

[LT12] Peter Lammich and Thomas Tuerk. “Applying Data Refinement for Monadic Programs to Hopcroft’s Algorithm”. In: *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*. Ed. by Lennart Beringer and Amy P. Felty. Vol. 7406. Lecture Notes in Computer Science. Springer, 2012, pp. 166–182. DOI: 10.1007/978-3-642-32347-8_12.

[Maz86] Antoni W. Mazurkiewicz. “Trace Theory”. In: *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 September 1986*. Ed. by Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg. Vol. 255. Lecture Notes in Computer Science. Springer, 1986, pp. 279–324. DOI: 10.1007/3-540-17906-2_30.

[Mer00] Stephan Merz. “Weak Alternating Automata in Isabelle/HOL”. In: *Theorem Proving in Higher Order Logics, 13th International Conference, TPHOLs 2000, Portland, Oregon, USA, August 14-18, 2000, Proceedings*. Ed. by Mark Aagaard and John Harrison. Vol. 1869. Lecture Notes in Computer Science. Springer, 2000, pp. 424–441. DOI: 10.1007/3-540-44659-1_26.

[Mer12] Stephan Merz. “Stuttering Equivalence”. In: *Archive of Formal Proofs* 2012 (2012). URL: https://www.isa-afp.org/entries/Stuttering_Equivalence.shtml.

[Mic88] Max Michel. “Complementation is more difficult with automata on infinite words”. Manuscript. 1988.

[Neu14] René Neumann. “Using Promela in a Fully Verified Executable LTL Model Checker”. In: *Verified Software: Theories, Tools and Experiments - 6th International Conference, VSTTE 2014, Vienna, Austria, July 17-18, 2014, Revised Selected Papers*. Ed. by Dimitra Giannakopoulou and Daniel Kroening. Vol. 8471. Lecture Notes in Computer Science. Springer, 2014, pp. 105–114. DOI: 10.1007/978-3-319-12154-3_7.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer, 2002. ISBN: 3-540-43376-7. DOI: 10.1007/3-540-45949-9.

[NTA96] Mohamed Naimi, Michel Tréhel, and André Arnold. “A Log(N) Distributed Mutual Exclusion Algorithm Based on Path Reversal”. In: *Journal of Parallel and Distributed Computing* 34.1 (1996), pp. 1–13. DOI: 10.1006/jpdc.1996.0041.

[NVW20] Thomas Neele, Antti Valmari, and Tim A. C. Willemse. “The Inconsistent Labelling Problem of Stutter-Preserving Partial-Order Reduction”. In: *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*. Ed. by Jean Goubault-Larrecq and Barbara König. Vol. 12077. Lecture Notes in Computer Science. Springer, 2020, pp. 482–501. DOI: 10.1007/978-3-030-45231-5_25.

[Pel93] Doron A. Peled. “All from One, One for All: on Model Checking Using Representatives”. In: *Computer Aided Verification, 5th International Conference, CAV ’93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*. Ed. by Costas Courcoubetis. Vol. 697. Lecture Notes in Computer Science. Springer, 1993, pp. 409–423. DOI: 10.1007/3-540-56922-7_34.

[Pel96] Doron A. Peled. “Combining Partial Order Reductions with On-the-Fly Model-Checking”. In: *Formal Methods in System Design* 8.1 (1996), pp. 39–64. DOI: 10.1007/BF00121262.

[Pel98] Doron A. Peled. “Ten Years of Partial Order Reduction”. In: *Computer Aided Verification, 10th International Conference, CAV ’98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*. Ed. by Alan J. Hu and Moshe Y. Vardi. Vol. 1427. Lecture Notes in Computer Science. Springer, 1998, pp. 17–28. DOI: 10.1007/BFb0028727.

[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*. IEEE Computer Society, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[PW97] Doron A. Peled and Thomas Wilke. “Stutter-Invariant Temporal Properties are Expressible Without the Next-Time Operator”. In: *Information Processing Letters* 63.5 (1997), pp. 243–246. DOI: 10.1016/S0020-0190(97)00133-6.

[RS59] Michael O. Rabin and Dana S. Scott. “Finite Automata and Their Decision Problems”. In: *IBM Journal of Research and Development* 3.2 (1959), pp. 114–125. DOI: 10.1147/rd.32.0114.

[Sac+19] Robert Sachtleben, Robert M. Hierons, Wen-ling Huang, and Jan Peleska. “A Mechanised Proof of an Adaptive State Counting Algorithm”. In: *Testing Software and Systems - 31st IFIP WG 6.1 International Conference, ICTSS 2019, Paris, France, October 15-17, 2019, Proceedings*. Ed. by Christophe Gaston, Nikolai Kosmatov, and Pascale Le Gall. Vol. 11812. Lecture Notes in Computer Science. Springer, 2019, pp. 176–193. DOI: 10.1007/978-3-030-31280-0_11.

[Sac19] Robert Sachtleben. “Formalisation of an Adaptive State Counting Algorithm”. In: *Archive of Formal Proofs* 2019 (2019). URL: https://www.isa-afp.org/entries/Adaptive_State_Counting.html.

[Saf88] Shmuel Safra. “On the Complexity of omega-Automata”. In: *29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988*. IEEE Computer Society, 1988, pp. 319–327. DOI: 10.1109/SFCS.1988.21948.

[Sch09] Sven Schewe. “Büchi Complementation Made Tight”. In: *26th International Symposium on Theoretical Aspects of Computer Science, STACS 2009, February 26-28, 2009, Freiburg, Germany, Proceedings*. Ed. by Susanne Albers and Jean-Yves Marion. Vol. 3. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2009, pp. 661–672. DOI: 10.4230/LIPIcs.STACS.2009.1854.

[SE05] Stefan Schwoon and Javier Esparza. “A Note on On-the-Fly Verification Algorithms”. 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. Ed. by Nicolas Halbwachs and Lenore D. Zuck. Vol. 3440. Lecture Notes in Computer Science. Springer, 2005, pp. 174–190. DOI: 10.1007/978-3-540-31980-1_12.

[Sic+16] Salomon Sickert, Javier Esparza, Stefan Jaax, and Jan Kretínský. "Limit-Deterministic Büchi Automata for Linear Temporal Logic". In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Vol. 9780. Lecture Notes in Computer Science. Springer, 2016, pp. 312–332. DOI: 10.1007/978-3-319-41540-6_17.

[Sic15] Salomon Sickert. "Converting Linear Temporal Logic to Deterministic (Generalised) Rabin Automata". In: *Archive of Formal Proofs* 2015 (2015). URL: https://www.isa-afp.org/entries/LTL_to_DRA.shtml.

[Sic16] Salomon Sickert. "Linear Temporal Logic". In: *Archive of Formal Proofs* 2016 (2016). URL: https://www.isa-afp.org/entries/LTL.shtml.

[Sie12] Stephen F. Siegel. "Transparent partial order reduction". In: *Formal Methods in System Design* 40.1 (2012), pp. 1–19. DOI: 10.1007/s10703-011-0126-0.

[Sie19] Stephen F. Siegel. "What's Wrong with On-the-Fly Partial Order Reduction". In: *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*. Ed. by Isil Dillig and Serdar Tasiran. Vol. 11562. Lecture Notes in Computer Science. Springer, 2019, pp. 478–495. DOI: 10.1007/978-3-030-25543-5_27.

[SMS09] Alexander Schimpf, Stephan Merz, and Jan-Georg Smaus. "Construction of Büchi Automata for LTL Model Checking Verified in Isabelle/HOL". In: *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*. Ed. by Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel. Vol. 5674. Lecture Notes in Computer Science. Springer, 2009, pp. 424–439. DOI: 10.1007/978-3-642-03359-9_29.

[SS19] Benedikt Seidl and Salomon Sickert. "A Compositional and Unified Translation of LTL into ω-Automata". In: *Archive of Formal Proofs* 2019 (2019). URL: https://www.isa-afp.org/entries/LTL_Master_Theorem.html.

[SS78] William J. Sakoda and Michael Sipser. “Nondeterminism and the Size of Two Way Finite Automata”. In: *Proceedings of the 10th Annual ACM Symposium on Theory of Computing, May 1-3, 1978, San Diego, California, USA*. Ed. by Richard J. Lipton, Walter A. Burkhard, Walter J. Savitch, Emily P. Friedman, and Alfred V. Aho. ACM, 1978, pp. 275–286. DOI: 10.1145/800133.804357.

[SVW87] A. Prasad Sistla, Moshe Y. Vardi, and Pierre Wolper. “The Complementation Problem for Büchi Automata with Appplications to Temporal Logic”. In: *Theoretical Computer Science* 49 (1987), pp. 217–237. DOI: 10.1016/0304-3975(87)90008-9.

[Tsa+07] Yih-Kuen Tsay, Yu-Fang Chen, Ming-Hsien Tsai, Kang-Nien Wu, and Wen-Chin Chan. “GOAL: A Graphical Tool for Manipulating Büchi Automata and Temporal Formulae”. 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*. Ed. by Orna Grumberg and Michael Huth. Vol. 4424. Lecture Notes in Computer Science. Springer, 2007, pp. 466–471. DOI: 10.1007/978-3-540-71209-1_35.

[Tsa+14] Ming-Hsien Tsai, Seth Fogarty, Moshe Y. Vardi, and Yih-Kuen Tsay. “State of Büchi Complementation”. In: *Logical Methods in Computer Science* 10.4 (2014). DOI: 10.2168/LMCS-10(4:13)2014.

[Val89] Antti Valmari. “Stubborn sets for reduced state space generation”. In: *Advances in Petri Nets 1990 [10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany, June 1989, Proceedings]*. Ed. by Grzegorz Rozenberg. Vol. 483. Lecture Notes in Computer Science. Springer, 1989, pp. 491–515. DOI: 10.1007/3-540-53863-1_36.

[Var07] Moshe Y. Vardi. “The Büchi Complementation Saga”. In: *STACS 2007, 24th Annual Symposium on Theoretical Aspects of Computer Science, Aachen, Germany, February 22-24, 2007, Proceedings*. Ed. by Wolfgang Thomas and Pascal Weil. Vol. 4393. Lecture Notes in Computer Science. Springer, 2007, pp. 12–22. DOI: 10.1007/978-3-540-70918-3_2.

[Var95] Moshe Y. Vardi. “An Automata-Theoretic Approach to Linear Temporal Logic”. In: *Logics for Concurrency - Structure versus Automata (8th Banff Higher Order Workshop, Banff, Canada, August 27 - September 3, 1995, Proceedings)*. Ed. by Faron Moller and

Graham M. Birtwistle. Vol. 1043. Lecture Notes in Computer Science. Springer, 1995, pp. 238–266. DOI: 10.1007/3-540-60915-6_6.

[VW86] Moshe Y. Vardi and Pierre Wolper. "An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report)". In: *Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986*. IEEE Computer Society, 1986, pp. 332–344.

[Wad92] Philip Wadler. "Comprehending Monads". In: *Mathematical Structures in Computer Science* 2.4 (1992), pp. 461–493. DOI: 10.1017/S0960129500001560.