BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

AN AUTOMATA-BASED DECISION PROCEDURE

ROZHODOVACÍ PROCEDURA ZALOŽENÁ NA AUTOMATECH

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTOR **MICHAL HEČKO**
AUTOR PRÁCE

SUPERVISOR **Ing. ONDŘEJ LENGÁL, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2022

Department of Intelligent Systems (DITS) Academic year 2021/2022

Bachelor's Thesis Specification

24744

Student:	**Hečko Michal**
Programme:	Information Technology
Title:	**An Automata-Based Decision Procedure**
Category:	Theoretical Computer Science

Assignment:

1. Study the theory of finite automata and efficient algorithms for manipulating them.
2. Study automata-based decision procedures for selected first or second-order theories, such as linear integer arithmetic, (W)S1S, separation logic, etc.
3. Analyse the bottlenecks of the studies procedure(s) and develop efficient algorithms and/or encodings for dealing with those bottlenecks. Use real-world benchmarks, e.g., from the SMT Competition, to guide you. Focus on formulae with quantifier alternations.
4. Implement the developed algorithms.
5. Evaluate the performance of the improved algorithms and compare it with (a) basic automata-based decision procedure for the logic and (b) state-of-the-art solvers (which might be based on a different technique).
6. Discuss the achieved results and propose directions for future work.

Recommended literature:

- B. Boigelot and P. Wolper, "Representing Arithmetic Constraints with Finite Automata: An Overview," in *Logic Programming*, vol. 2401, P. J. Stuckey, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 1-20.
- A. Durand-Gasselin and P. Habermehl, "On the Use of Non-deterministic Automata for Presburger Arithmetic," in *CONCUR 2010 - Concurrency Theory*, vol. 6269, P. Gastin and F. Laroussinie, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 373-387.
- Tomáš Fiedor, Lukáš Holík, Petr Janků, Ondřej Lengál, Tomáš Vojnar: Lazy Automata Techniques for WS1S. TACAS (1) 2017: 407-425

Requirements for the first semester:

- Points (1) and (2).

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Lengál Ondřej, Ing., Ph.D.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	November 1, 2021
Submission deadline:	May 11, 2022
Approval date:	November 3, 2021

Abstract

Presburger arithmetics (PrA) is a decidable, first-order theory of natural numbers, with applications in many areas in formal verification of software properties. SMT-solvers — tools implementing various algorithmic approaches to deciding whether a formula has a solution — play a crucial role in formal verification. In this work, we document building a novel automatic SMT solver for PrA based on finite automata — an approach that no SMT solver currently employs. We provide an overview of challenges and their solutions arising from the complexity of such a tool, including results from the conducted experiments already showing problems in which this alternative approach outperforms the state-of-the-art solvers. We have also identified problems in which the performance of the automata-based procedure struggles, which are open research opportunities.

Abstrakt

Presburgerova aritmetika (PrA) je rozhodnutelná teorie přirozených čísel prvního řádu, která nachází uplatnění v mnoha oblastech formální verifikace vlastností softwaru. Řešiče SMT — nástroje implementující různé algoritmické přístupy k rozhodování, zda má formule řešení — hrají ve formální verifikaci klíčovou roli. V této práci dokumentujeme vytvoření nového automatického SMT řešiče pro PrA založeného na konečných automatech — přístupu, který v současnosti žádný SMT řešič nepoužívá. Uvádíme přehled výzev a jejich řešení vyplývajících ze složitosti takového nástroje, včetně výsledků z provedených experimentů, které již identifikují problémy, kde tento alternativní přístup překonává nejmodernější řešiče. Uvádíme také identifikované problémy, u nichž výkonnost postupu založeného na automatech naráží na problémy, které představují otevřené možnosti výzkumu.

Keywords

Presburger arithmetic, SMT solver, Linear integer arithmetic, Finite automaton

Klíčová slova

Presburgerova aritmetika, SMT solver, Celočíselná lineárna aritmetika, Konečný automat

Reference

HEČKO, Michal. *An Automata-Based Decision Procedure.* Brno, 2022. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Ondřej Lengál, Ph.D.

Rozšířený abstrakt

Rostoucí využívání počítačů téměř ve všech oblastech lidského zájmu vyvolává potřebu bezpečného a spolehlivého softwaru, což přivádí pozornost k úloze formálního ověřování vlastností softwaru. Jedním ze základních nástrojů používaných při formální verifikaci je řešič SMT (Satisfiability Modulo Theories) — nástroj schopný automaticky odvodit, zda daná formule prvního řádu popisující formálně systém má model, což znamená, že systém má požadovanou vlastnost. Jednou z dominantních teorií používaných ve vstupních for- mulích je Presburgerova aritmetika (PrA), která je schopna popsat systém pomocí lineárních celočíselných omezení, jako jsou rovnice a nerovnice. Protože PrA neobsahuje notaci ani axiomy pro násobení, je její vyjadřovací schopnost omezená. Omezená vyjadřovací síla však umožňuje, aby teorie zůstala rozhodnutelná, což znamená, že je možné určit existenci řešení algoritmicky v konečném počtu kroků. V průběhu času byla vyvinuta řada algoritmů — rozhodovacích procedur — přistupujících k problému určení existence řešení z různých pohledů. Jeden z mladších přístupů je založen na formálním modelu konečných automatů, v němž je pro každé atomické omezení, např. nerovnici nebo rovnici, konstruován konečný automat. Tyto automaty se pak kombinují podle struktury vstupní formule, čímž vzniká automat kódující celý prostor řešení formule. Vstupní formule má tedy model, jestliže automat kódující celý prostor řešení má neprázdný jazyk. Pokud je nám známo, nebyla dosud v žádném řešiči SMT implementována rozhodovací procedura založená na konečných automatech. Proto nebyly pokroky v oblasti automatů vyhodnoceny v kontextu rozhodování PrA, ani není možné porovnat postup založený na automatech s nejmodernějšími řešiči SMT a určit oblasti, kde by automaty mohly představovat výkonnější přístup.

V této práci dokumentujeme implementaci Amaya - nového experimentálního automatového SMT řešiče pro PrA, čímž pokládáme nezbytné základy pro budoucí výzkum automatového přístupu k rozhodování PrA. Protože primárním účelem Amaya je experimentování s procedurou založenou na automatech, odrážejí tento účel i funkce, které Amaya poskytuje. Tyto funkce sahají od úplné introspekce do rozhodovací procedury prostřednictvím graficky zobrazitelného výstupu všech mezivýsledných automatů vytvořených během běhu procedury až po poskytování pokročilého vestavěného mechanismu pro benchmarking. Amaya také podporuje standardizovaný vstupní jazyk, což minimalizuje úsilí potřebné při porovnávání základní rozhodovací procedury s přístupy implementovanými jinými řešiči. Identifikovali jsme také problémy se škálovatelností klasických automatových konstrukcí způsobené exponenciálním růstem jejich časové složitosti vzhledem k počtu proměnných ve vstupní formuli a vyřešili je návrhem reprezentace přechodové relace automatů založené na víceterminálních binárních rozhodovacích diagramech (MTBDD). Abychom plně využili výhod MTBDD, museli jsme přeformulovat všechny klasické automatové algoritmy tak, aby využívaly kompaktní reprezentaci, kterou MTBDD poskytují. Amaya tedy poskytuje dva prováděcí backendy: nativní backend zaměřený na experimentování, který reprezentuje přechody explicitně, a backend založený na MTBDD zaměřený na výkon. Amayu jsme také porovnali s nejmodernějšími řešiči Z3 a CVC5 při rozhodování benchmarků, které mají původ ve verifikaci programů. Provedené experimenty navíc již identifikovaly oblasti, jako je rozhodování Frobeniova problému, kde automaty představují výrazně výkonnější přístup než přístupy implementované Z3 a CVC5. Naše experimenty také ukázaly oblasti, kde je postup založený na automatech neproveditelně drahý, což poskytuje otevřené možnosti výzkumu.

An Automata-Based Decision Procedure

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Ondřej Lengál. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

....
Michal Hečko
May 9, 2022

Acknowledgements

I would like to thank my supervisor Ing. Ondřej Lengál, Ph.D., for his patience and guidance. I also wish to express my thanks to doc. Mgr. Lukáš Holík, Ph.D. and Ing. Vojtěch Havlena, Ph.D. for providing me with inspiration and many constructive ideas. Finally, my gratitude belongs to Michaela Mesárošová for her tremendous support.

Contents

Chapter 1

Introduction

As humanity advances, it increasingly relies on computers to perform some of the tasks previously requiring human intellect. Likewise, challenging engineering undertakings set forth by modern society in various areas of human interest, such as space exploration, are deemed unfeasible without the employment of computers. The growing penetration of information technologies into everyday life creates the necessity for safe and reliable software controlling those devices, bringing attention to the role of formal verification of software properties.

One of the core tools used in software and hardware verification is a Satisfiability Modulo Theories (SMT) solver — a tool capable of automatically deducing the existence of a model of a given first-order formula. An SMT solver is a versatile tool, and its applications can be found in various other areas of modern computer science, such as compiler optimization techniques or automated theorem proving, further highlighting the importance of this tool.

One of the prominent theories used in the input formulae of SMT solvers is Presburger Arithmetic (PrA). This first-order theory of integers provides a formal basis for describing a system in terms of linear arithmetic constraints. The signature of PrA is similar to that of the well-known Peano Arithmetic [23], but it does not contain a symbol nor the corresponding axioms for multiplication. Although this limitation significantly reduces its expressive power, it also allows the theory to remain decidable, meaning that it is possible to deduce whether a PrA formula has a model algorithmically in a finite number of steps. Over time there have been numerous such algorithms — *decision procedures* — developed, approaching the problem of determining the existence of a model from different perspectives. A lot of current research, e.g., [17], [8], focuses on developing various heuristics that improve the performance of these procedures, extending what we can decide automatically.

The most used decision procedure for PrA is *quantifier elimination*. As the name suggests, the procedure systematically transforms an input formula with quantifiers into an equivalent quantifier-free formula. The procedure has a long history reaching all the way to the work of Mojżesz Presburger, who developed this algorithm to show the decidability of PrA [24]. Given the age of this procedure, combined with its popularity, there have been numerous efforts analyzing its characteristics and developing optimizations for specific types of PrA formulae.

A younger approach to determining the existence of a model is based on the formal model of finite automata (FAs). Instead of gradually transforming the input formula into a quantifier-free equivalent, the procedure compactly represents the entire solution space of linear constraints, e.g. inequations present in the input formula using automata. The procedure then combines the automata corresponding to linear constraints according to the structure of the input formula, using automata constructions equivalent to the logical conjunctions in the input formula. The idea behind an automata-based decision procedure originates in the works of Büchi [7], but it was not until 1996 when Boudet and Comon applied it in the context of PrA [5]. The procedure's underlying formal model is frequently applied throughout the modern computer science, e.g., in text processing, or to detect attacks in network traffic, etc., and, therefore, the automata theory is a vivid research area with advancements such as new automata models [26] or improved algorithms for checking universality of the language [1]. To the best of our knowledge, no decision procedure for PrA based on this approach has been implemented in any SMT solver yet.

Furthermore, there are known problems, such as the Frobenius coin problem at which automata-based procedure vastly outperforms the approaches used by the state-of-the-art SMT solvers. The coin problem is a famous mathematical problem that can be formulated in the following manner: given an unlimited number of coins of certain denominations, what is the highest possible sum one cannot achieve by any combination of the given coins? The coin problem is not only related to currency, but its applications are found in other areas such as the analysis of P/T systems [9]. In order to answer the question of whether there are other problems in which the automata-based approach provides advantages compared to the other decision procedures cannot by answered without an SMT solver implementing the automata-decision procedure.

In our work, we have succeeded in building an automata-based SMT solver for PrA, laying a foundation for the future research of the decision procedure. The solver was designed to support a standardized input language, enabling easy comparison of the automata-based procedure to the alternative approaches implemented by the state-of-the-art SMT solvers. While developing the automata-based solver, we have identified and solved numerous engineering challenges ranging from developing the necessary algorithm to repair automata after modifying them as a consequence of an existential quantifier in the input formula to developing a high-performance representation of the automaton transition relation based on Multi-Terminal Binary Decision Diagrams. As the purpose of the created tool is to provide a basis for future research, the solver is capable of outputting the automata created throughout the decision procedure to various formats such as the DOT language, allowing a graphical introspection of the decision procedure. We have conducted experiments comparing our solver to the state-of-the-art solvers Z3 and CVC5 in some of the available benchmarks. Furthermore, we confirmed the superior performance of the automata-based decision procedure in deciding the Frobenius coin problem. The conducted experiments already identified performance problems of the implemented approach caused by the sizes of the automata for linear constraints, presenting an attractive research topic of representing parts of the state space symbolically.

Chapter 2

Preliminaries

In this chapter, we introduce all necessary definitions used throughout the remainder of this work. We start with a description of the Presburger arithmetic, its properties, and its relation to regular languages. We continue with the definition of finite automata, the class of languages they encode, and the properties of those languages, including corresponding algorithms providing the foundations for the implementation of the automata-based decision procedure. Lastly, we introduce Binary decision diagrams (BDDs)—an efficient representation of Boolean functions—and their extension named Multi-terminal binary decision diagrams (MTBDDs), allowing compact representation of functions from multi-dimensional Boolean space to any finite set. Such representations will become vital for the performance of the automata-based decision procedure, as they provide an effective way to address the scalability issues of classical automata constructions wrt. the number of variables in the input formula.

Throughout this text, we use the following notation:

- lower-case letters $a, b, c...$ denote constants,
- lower-case letters $x, y, z...$ denote first-order variables,
- Greek letters $\varphi, \psi, ...$ denote (first-order) formulae.

2.1 Presburger arithmetic

Presburger arithmetic is a first-order theory of the natural numbers first studied in detail by Mojżesz Presburger in 1929. It was at a time when the movement initiated by David Hilbert (the so-called Hilbert's program) was in search of a formal system providing solid foundations for the entirety of mathematics. This system would rely on mathematical logic as a language preventing all ambiguities and providing means to formalize and manipulate mathematical statements. Hilbert also set forth the properties such a system must exhibit, among which was decidability—the ability to tell whether a given statement is true or false in a finite number of steps. Presburger arithmetic (PrA), being much weaker than Peano arithmetic [23] was proven by Presbuger to have all of the required properties, and therefore, it has been seen as a step to achieving the goals of Hilbert's program. In 1931 Gödel published his results [16] that proved the vision of a formal system having all of the properties proposed by Hilbert impossible to achieve. However, PrA remained interesting, especially due to its tractability by automated reasoning.

PrA is a first-order theory with equality and the following language:

1. A constant symbol 0,
2. a unary function symbol S, called *successor*,
3. a binary function symbol $+$, called addition.

PrA contains the following axioms:

1. $\forall x \neg(S(x) = 0)$
2. $\forall x, y(S(x) = S(y) \rightarrow x = y)$
3. $\forall x(x + 0 = x)$
4. $\forall x, y(x + S(y) = S(x + y))$
5. (First-order) axiom schema of induction: For a formula φ with a single free variable x: $(\varphi(0) \wedge \forall x(\varphi(x) \rightarrow \varphi(S(x)))) \rightarrow \forall x(\varphi(x))$

The standard model of PrA are natural numbers (including 0) ordered as usual ($S(0) = 1$, $S(1) = 2$).

We also introduce the following abbreviation that allows us to write a *numeral* in the standard way:

$$n \overset{def}{=} \underbrace{S(S(\ldots S(0)))}_{n}$$

In contrast to Peano axioms, the theory does not include the symbol and axioms for multiplication; however, multiplication by a constant can be defined, as it is a shorthand for summation:

$$c \cdot x \overset{def}{=} \underbrace{x + x \cdots + x}_{c}$$

The total ordering $\leq$ can be defined as follows:

$$a \leq b \overset{def}{=} \exists c(a + c = b)$$

Both the integer division by a constant (denoted as x *div* c) and the modulo operation limited to a constant divisor (denoted as x *mod* c), can be defined in PrA as follows:

$$x \; div \; c = y \overset{def.}{\iff} \exists z(x - c \cdot y = z \wedge 0 \leq z \wedge z < c)$$
$$x \; mod \; c = y \overset{def.}{\iff} \exists z(x - c \cdot z = y \wedge 0 \leq y \wedge y < c)$$

Let $\vec{x}$ be a vector of $n \geq 0$ variables, $\vec{a} \in \mathbb{Z}^n$ be a vector of variable coefficients, $c \in \mathbb{Z}$, $m \in \mathbb{N}$ be constants. A formula $\varphi : \vec{a} \cdot \vec{x} \sim c$ is called *atomic*, where $\sim$ is $\leq$, $<$, $=$ or $\equiv_m$.

PrA has been shown to be complete and decidable by Mojżesz Presburger as a part of his master thesis [24].

2.2 Automata theory

This chapter introduces the definition of classical *finite automata* (FAs) and the class of languages this model of computation recognizes. As FAs pose a basis for the automata-based decision procedure, they will be often referred to throughout the rest of this text. We also introduce key operations and properties of languages recognized by FAs, including the algorithms performing language operations on automata.

Throughout this text, the following notation is employed. Let Σ be a finite non-empty set of symbols. Σ^* denotes the set of all words over Σ, Σ^+ is the set of all words over Σ except the empty string ε. Furthermore, $\mathcal{P}(\Sigma)$ denotes the set of all subsets of Σ (the powerset of Σ).

Definition 2.2.1. *A* nondeterministic finite automaton (NFA, FA) $\mathcal{A}$ *is a 5-tuple* $(Q, \Sigma, \delta, Q_0, F)$*, where* Q *is a finite non-empty set of states,* Σ *is a finite non-empty set of symbols called an alphabet,* $\delta \subseteq Q \times \Sigma \times Q$ *is the set of transitions,* $Q_0 \subseteq Q$ *is the set of initial states, and* $F \subseteq Q$ *is the set of final states.*

Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ be an automaton. A transition $(q, \sigma, q') \in \delta$ from the state $q \in Q$ to the state $q' \in Q$ labeled with the symbol $\sigma \in \Sigma$ is denoted by $q \xrightarrow{\sigma} q' \in \delta$. Let $w \in \Sigma^*$ be a word, $|w|$ denotes the length of this word and let w_i denote the i-th symbol of the word w for every $1 \leq i \leq |w|$. A *run* ρ of an automaton $\mathcal{A}$ over a word $w = w_1 w_2 \cdots w_n \in \Sigma^*$ from the state $p \in Q$ to the state $r \in Q$, denoted as $p \overset{w}{\Longrightarrow} r$, is a sequence of states $\rho = q_0, q_1, \cdots, q_n$ that satisfies $q_{i-1} \xrightarrow{w_i} q_i \in \delta$ for $1 \leq i \leq n$, $q_0 = p$, $q_{|w|} = r$. Furthermore, let $p \Longrightarrow r$ denote the existence of such a run. The run ρ is *accepting* iff $r \in F$. A state is called *unreachable* iff there exists no run from an initial state to the given state. Let $q \in Q$ be a state and $S \subseteq Q$ be a set of states. We define the following.

$$
\begin{aligned}
pre_\delta(q) &= \{q' \mid q' \xrightarrow{\sigma} q \in \delta\} \\
pre_\delta(S) &= \bigcup\nolimits_{q \in S} pre(q) \\
post_\delta(q) &= \{q' \mid q \xrightarrow{\sigma} q' \in \delta\} \\
post_\delta(S) &= \bigcup\nolimits_{q \in S} post(q)
\end{aligned}
$$

The *language* of the state q, denoted by $\mathcal{L}(q)$, is defined by $\mathcal{L}(q) = \{w \mid q \overset{w}{\Longrightarrow} q_f, q_f \in F\}$. A state that has an empty language is called *nonfinishing.* The language of the automaton $\mathcal{A}$, denoted as $\mathcal{L}(\mathcal{A})$, is defined by $\mathcal{L}(\mathcal{A}) = \bigcup_{q_0 \in Q_0} \mathcal{L}(q_0)$. A language is called *regular* iff it is recognized by some finite automaton.

A *deterministic finite automaton (DFA)* is an FA that has only one initial state ($|Q_0| = 1$) and its structure satisfies $\forall q \in Q(\forall \sigma \in \Sigma(\exists! q' \in Q(q \xrightarrow{\sigma} q' \in \delta)))$. A DFA is called *complete* iff for every $w \in \Sigma^*$ there exists a run of the automaton from some initial state $q_0 \in Q_0$ over w.

Theorem 2.2.1. *The class of regular languages is closed under the intersection operation.*

Proof. Let $\mathcal{A}_1 = (Q_1, \Sigma_1, \delta_1, Q_{01}, F_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma_2, \delta_2, Q_{02}, F_2)$ be two NFAs with $Q_1 \cap Q_2 = \emptyset$. Then the following construction produces an automaton $\mathcal{A}_\cap = (Q, \Sigma, \delta, Q_0, F)$ such that $\mathcal{L}(\mathcal{A}_\cap) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$:

- $Q = Q_1 \times Q_2$,
- $\Sigma = \Sigma_1 \cap \Sigma_2$,
- $\delta = \{(q_1, 1_2) \xrightarrow{\sigma} (q_1', q_2') \mid q_1 \xrightarrow{\sigma}, q_1' \in \delta_1 \wedge q_2 \xrightarrow{\sigma}, q_2' \in \delta_2\}$,
- $Q_0 = Q_{01} \times Q_{02}$, and
- $F = F_1 \times F_2$.

Let $w = w_1 \cdots w_n$ be a word and $w \in \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$. Such word must be accepted by both $\mathcal{A}_1$ and $\mathcal{A}_2$. Let $\rho_1 = q_0 \ldots q_n$ and $\rho_2 = s_0 \ldots s_n$ be the runs of $\mathcal{A}_1$, $\mathcal{A}_2$ over w, respectively. As the states $q_0 \in Q_{01}$ and $s_0 \in Q_{02}$ are initial in corresponding automata, then by the construction of $\mathcal{A}_\cap$ there is an initial state $(q_0, q_1) \in Q_0$. Following the construction, let q_i and s_i be states of ρ_0, ρ_1, respectively for some $0 \leq i < n$. Then by construction there is a transition $(q_i, s_i) \xrightarrow{w_i} (q_{i+1}, s_{i+1}) \in \delta$ for every pair (q_i, s_i), and therefore, a run $\hat{\rho}$ of $\mathcal{A}_\cap$ over w exists. In order for w to be accepted by both $\mathcal{A}_1$ and $\mathcal{A}_2$, both q_n and s_n must be final. From the construction then follows that the $(q_n, s_n) \in F$, and therefore, $\hat{\rho}$ is an accepting run. We conclude that $(\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)) \subseteq \mathcal{L}(\mathcal{A}_\cap)$.

Let $v = v_1 \ldots v_n$ be a word, $v \in \mathcal{L}(\mathcal{A}_\cap)$ and let $\rho_v = (q_0, s_0) \ldots (q_0, s_0)$ be the corresponding run of $\mathcal{L}(\mathcal{A}_\cap)$ over v. By construction there is a transition $q_i \xrightarrow{v_{i+1}} q_{i+1}$ of automaton $\mathcal{A}_1$ for $0 \leq i < n$. As (q_n, s_n) is an accepting state, the states q_n, s_n must be also accepting in $\mathcal{A}_1$ and $\mathcal{A}_1$, respectively. From the construction follows that q_0 must be an initial state of $\mathcal{A}_1$. Therefore there exitsts an accepting run $\rho_1\colon q_0 \overset{v}{\Longrightarrow} q_n$ of $\mathcal{A}_1$. Similarly, there must exist an accepting run $\rho_2\colon s_0 \overset{v}{\Longrightarrow} s_n$ of $\mathcal{A}_2$. Therefore, $\forall v \in \mathcal{L}(\mathcal{A}_\cap)(v \in \mathcal{L}(\mathcal{A}_1) \wedge \mathcal{L}(\mathcal{A}_2))$, from which we conclude $\mathcal{L}(\mathcal{A}_\cap) \subseteq \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$.

As $(\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)) \subseteq \mathcal{L}(\mathcal{A}_\cap)$ and $\mathcal{L}(\mathcal{A}_\cap) \subseteq (\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2))$, the language of automaton $\mathcal{A}_\cap$ is precisely $(\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)) = \mathcal{L}(\mathcal{A}_\cap)$, and therefore, the class of regular languages is closed under the intersection operation.

□

Intuitively, the depicted construction creates an automaton with states being tuples capturing every possible combination of states from $\mathcal{A}_1$ and $\mathcal{A}_2$. Transitions are added between these states only if both $\mathcal{A}_1$ and $\mathcal{A}_2$ could make a transition via some symbol σ to respective states. However, such construction produces an automaton with unnecessary states — either unreachable or nonfinishing. For practical use, it is better to apply the procedure *NFAIntersection* 1, which avoids the creation of unreachable states by working incrementally, adding states only when their reachability is certain.

Algorithm 1 Construction of an automaton recognizing the intersection of two regular languages

Input: NFAs $\mathcal{A}_1 = (Q_1, \Sigma_1, \delta_1, Q_{01}, F_1), \mathcal{A}_2 = (Q_2, \Sigma_2, \delta_2, Q_{02}, F_2)$
Output: NFA $\mathcal{A}_\cap = (Q, \Sigma, \delta, Q, F)$ such that $\mathcal{L}(\mathcal{A}_\cap) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$

```
1: function NFAIntersection($\mathcal{A}_1, \mathcal{A}_2$)
2:     $Q, \delta, F \leftarrow \emptyset$
3:     $\Sigma \leftarrow \Sigma_1 \cap \Sigma_2$
4:     $Q_0 \leftarrow Q_{01} \times Q_{02}$
5:     $W \leftarrow Q_{01} \times Q_{02}$
6:     while $W \neq \emptyset$ do
7:         $(q_1, q_2) \leftarrow$ pick and remove state from $W$
```

```
            add $(q_1, q_2)$ to $Q$
            if $q_1 \in F_1 \wedge q_2 \in F_2$ then
                add $(q_1, q_2)$ to $F$
            end if
            for each $\sigma \in \Sigma$ do
                for each $q_1' \in \{q_1' \mid q_1 \xrightarrow{\sigma} q_1' \in \delta_1\}$ and each $q_2' \in \{q_2' \mid q_2 \xrightarrow{\sigma} q_2' \in \delta_2\}$ do
                    add $(q_1, q_2) \xrightarrow{\sigma} (q_1', q_2')$ to $\delta$
                    if $(q_1', q_2') \notin Q$ then
                        add $(q_1', q_2')$ to $W$
                    end if
                end for
            end for
        end while
        return $(Q, \Sigma, \delta, Q_0, F)$
    end function
```

Theorem 2.2.2. *The class of regular languages is closed under the union operation.*

Proof. Let $\mathcal{A}_1 = (Q_1, \Sigma_1, \delta_1, Q_{01}, F_1)$, $\mathcal{A}_2 = (Q_2, \Sigma_2, \delta_2, Q_{02}, F_2)$ be two NFAs with $Q_1 \cap Q_2 = \emptyset$. Then the following NFA $\mathcal{A}_\cup = (Q_1 \cup Q_2, \Sigma_1 \cup \Sigma_2, \delta_1 \cup \delta_2, Q_1 \cup Q_2, F_1 \cup F_2)$ recognizes the language $\mathcal{L}(\mathcal{A}_\cup) = \mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$.

Let $w = w_1 \dots w_n$ be a word such that $w \in \mathcal{L}(\mathcal{A}_1)$, and let $\rho = q_0 \dots q_n$ be a run of $\mathcal{A}_1$ over w. From the construction of $\mathcal{A}_\cup$ we can see that all states of ρ are present in $\mathcal{A}_\cup$, and the same holds for the transitions between the states. The state $q_0 \in Q_{01}$ is also an initial state in $\mathcal{A}_\cup$, and $q_n \in F_1$ will be also one of the final states of $\mathcal{A}_\cup$. Therefore, there must be an accepting run $\hat{\rho}$ of $\mathcal{A}_\cup$ over w. Following a similar logic, the same conclusions can be reached for some $v \in \mathcal{L}(\mathcal{A}_2)$. We conclude that $\forall w \in \mathcal{L}(\mathcal{A}_1)(w \in \mathcal{L}(\mathcal{A}_\cup)) \wedge \forall w \in \mathcal{L}(\mathcal{A}_2)(w \in \mathcal{L}(\mathcal{A}_\cup))$, and therefore, $\mathcal{L}(\mathcal{A}_\cup) \subseteq (\mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2))$.

Let $u = u_1 \dots u_n$ be a word such that $u \in \mathcal{L}(\mathcal{A}_\cup)$, and let $\rho_u = r_0 \dots r_n$ be a run of $\mathcal{A}_\cup$ over u. The state r_0 must be initial in either $\mathcal{A}_1$, or $\mathcal{A}_2$. As we put forth a requirement that $Q_1 \cap Q_2 = \emptyset$, all of the states of ρ_u must belong to either $\mathcal{A}_1$ or $\mathcal{A}_2$. Let this automaton be denoted as $\hat{\mathcal{A}}$. From the construction we can see that the same transitions between the states of ρ_u must be also present in $\hat{\mathcal{A}}$. Same holds for the state r_n—as it is accepting in $\mathcal{A}_\cup$, it must be also accepting in $\hat{\mathcal{A}}$. Therefore, there must be an accepting run of $\hat{\mathcal{A}}$ over u, from which we conclude that $\forall w \in \mathcal{L}(\mathcal{A}_\cup)(w \in \mathcal{A}_1 \vee w \in \mathcal{A}_2)$ which is equivalent to $\mathcal{L}(\mathcal{A}_\cup) \subseteq (\mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2))$. □

Theorem 2.2.3. *For every NFA $\mathcal{A}$, there exits an equivalent DFA $\mathcal{A}^\mathcal{D}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}^\mathcal{D})$.*

Proof. Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ be an NFA. We construct a deterministic automaton $\mathcal{A}^\mathcal{D} = (\mathcal{Q}, \Sigma, \Delta, \mathcal{Q}_0, \mathcal{F})$, that satisfies $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}^\mathcal{D})$ as follows:

- $\mathcal{Q} = 2^Q$,
- $\Delta = \{S \xrightarrow{\sigma} S' \mid S \in \mathcal{P}(Q) \wedge \sigma \in \Sigma \wedge S' = \{s' \mid s \xrightarrow{\sigma} s', s \in S\}\}$,
- $\mathcal{Q}_0 = \{Q_0\}$, and
- $\mathcal{F} = \{S \in \mathcal{P}(Q) \mid S \cap F \neq \emptyset\}$.

Let $w = w_1 \dots w_n$ be a word such that $w \in \mathcal{L}(\mathcal{A})$, and let $\rho_w = q_0 \dots q_f$ be the corresponding accepting run of $\mathcal{A}$ over w. Let q_i, q_{i+1} be any two consequent states of ρ_w, where $0 \leq i < n$, and let w_{i+1} be the symbol of w such that $q_i \xrightarrow{w_{i+1}} q_{i+1}$. Clearly, for any $S_i \in \mathcal{Q}$, such that $q_i \in S$, there must exist $S_{i+1} \in \mathcal{Q}$, such that $S_i \xrightarrow{w_{i+1}} S_{i+1}$ and $q_{i+1} \in S_{i+1}$. From the construction we know that $\mathcal{A}^{\mathcal{D}}$ has only one initial state S_0, such that $q_0 \in S_0$. Therefore, there exists a run $\hat{\rho}_w = S_0 \dots S_n$ of $\mathcal{A}^{\mathcal{D}}$ over w, such that $q_i \in S_i$ for any $0 \leq i < n$. As ρ_w is accepting, then $q_n \in F$, and therefore, $S_n \in \mathcal{F}$, and thus, $\hat{\rho}_w$ is also accepting state. From this we conclude that $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}^{\mathcal{D}})$.

Let $u = u_1 \dots u_n$ be a word such that $w \in \mathcal{L}(\mathcal{A})$. Let $\rho_u = S_0 \dots S_n$ be the run of $\mathcal{A}^{\mathcal{D}}$ over u. As ρ_u is accepting, there must be a state $q_n \in S_n$, such that $q_n \in F$. From the definition of a run, there exists a transition $S_{n-1} \xrightarrow{u_n} S_n \in \Delta$. Following the construction of $\mathcal{A}^{\mathcal{D}}$, there is a state $q_{n-1} \in S_{n-1}$ such that $q_{n-1} \xrightarrow{u_n} q_n \in \delta$. Extending this logic for every pair of consequent states in ρ_u, starting from the last pair to the first pair, we arrive at a sequence of states $q_0 \dots q_n$. As $q_0 \in S_0$ the state q_0 must also be an initial state of $\mathcal{A}$. Given the way we constructed the sequence of states $q_0 \dots q_n$, and the fact that $q_0 \in Q_0$ and $q_n \in F$ we can conclude that for any word $u \in \mathcal{L}(\mathcal{A}^{\mathcal{D}})$ there exists an accepting run of $\mathcal{A}$, and therefore, $\mathcal{L}(\mathcal{A}^{\mathcal{D}}) \subseteq \mathcal{L}(\mathcal{A})$.

As $\mathcal{L}(\mathcal{A}^{\mathcal{D}}) \subseteq \mathcal{L}(\mathcal{A})$ and $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}^{\mathcal{D}})$, the language of the automaton $\mathcal{A}^{\mathcal{D}}$ is precisely $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}^{\mathcal{D}})$, and therefore, $\mathcal{A}$ and $\mathcal{A}^{\mathcal{D}}$ are equivalent.

□

Although DFAs and NFAs have an equal expressive power, NFAs can be exponentially smaller than their deterministic counterpart [13]. However, it is difficult to create an automaton accepting the complement of a language accepted by an NFA, compared to creating such an automaton from a DFA. Furthermore, NFAs do not have a canonical form.

Similarly as with language intersection, the construction used in the proof 2.2 creates automata that often have prohibitively too many states to be used in practice. Many of the created states are unreachable or nonfinishing. Algorithm 2, commonly referred to as the standard subset construction, produces a DFA without unreachable states. This is due to its incremental design—new DFA states are added only after they can be reached from some of the states already present in the DFA being constructed.

Algorithm 2 Determinization procedure

```
Input: NFA $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$
Output: DFA $\mathcal{A}^{\mathcal{D}} = (\mathcal{Q}, \Sigma, \Delta, \mathcal{Q}_0, \mathcal{F}), such that \mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}^{\mathcal{D}})$
 1: $W, \mathcal{Q}_0 \leftarrow \{Q_0\}$
 2: $\mathcal{Q}, \mathcal{F}, \Delta \leftarrow \emptyset$
 3: while $W \neq \emptyset$ do
 4:     $S \leftarrow$ pick and remove state from $W$
 5:     add $S$ to $\mathcal{Q}$
 6:     if $S \cap F \neq \emptyset$ then
 7:         add $S$ to $\mathcal{F}$
 8:     end if
 9:     for each $\sigma \in \Sigma$ do
10:         $S' \leftarrow \delta(S, \sigma)$
11:         if $S' \notin \mathcal{Q}$ then
12:             add $S'$ to $W$
13:         end if
```

```
14:         add $(S, \sigma, S')$ to $\Delta$
15:     end for
16: end while
17: return $(\mathcal{Q}, \Sigma, \Delta, \mathcal{Q}_0, \mathcal{F})$
```

Theorem 2.2.4. *The class of regular languages is closed under complement.*

Proof. Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ be a complete DFA. Then the DFA $\overline{\mathcal{A}} = (Q, \Sigma, \delta, Q_0, Q \setminus F)$ recognizes the language $\overline{\mathcal{L}(\mathcal{A})} = \Sigma^* \setminus \mathcal{L}(\mathcal{A})$. □

The construction of $\overline{\mathcal{A}}$ recognizing the language complement originates in the observation that, in a given complete DFA, there is exactly one run $q_0 \overset{w}{\Longrightarrow} r$ for any $w \in \Sigma^*$, where q_0 is the initial state. The word w belongs to the language $\mathcal{L}(\mathcal{A})$ iff $r \in F$. Therefore, by swapping accepting and non-accepting states, the word that was previously accepted is not and vice versa, concluding that $\mathcal{L}(\overline{\mathcal{A}}) = \overline{\mathcal{L}(\mathcal{A})}$.

2.3 Binary Decision Diagrams and Multi-terminal Binary Decision Diagrams

Binary decision diagrams (BDDs) are a well-known representation of Boolean functions. BDDs were originally introduced by C. Y. Lee in 1958 [21]. However, it was after the publishement of Bryant's work in 1986 [6] that BDDs started to gain popularity. By imposing new restrictions on the BDD structure, Bryant introduced a way how BDDs can compactly represent Boolean functions, as well as algorithms allowing the manipulation of such representations without the need of unpacking them first. Since then, they have found their application in numerous areas of modern computer science, especially in formal verification and computer-aided design (circuit synthesis). In this chapter, we define BDDs and their properties. After defining BDDs, we introduce an extension of BDDs called Multi-terminal binary decision diagrams (MTBDDs), allowing for a compact representation of functions from a multi-dimensional Boolean space to an arbitrary non-empty set.

Let $\mathbb{B} = \{0, 1\}$, let $f(v_1, \ldots, v_n) : \mathbb{B}^n \to \mathbb{B}$ be a Boolean function, let $V = \{v_1, \ldots, v_n\}$ be the set of the input variables of f, and let $\prec$ be a total ordering over V.

Definition 2.3.1. *BDD is a rooted directed acyclic graph formally represented as a 7-tuple $\mathcal{B} = (N, T, var, low, high, root, val)$, where:*

- *N is a set of nonterminals i.e. internal decision nodes,*
- *T is a set of terminals (leaves),*
- *$var \colon N \to V$ is a function labeling the nonterminals with the input variables of f,*
- *$low \colon N \to N \cup T$ is the low successor of nonterminals $n \in N$ for the value of the variable $var(n)$ being 0,*
- *$high \colon N \to N \cup T$ is the high successor of nonterminals $n \in N$ for the value of the variable $var(n)$ being 1,*
- *$root \in N \cup T$ is the root node,*
- *$val \colon T \to \mathbb{B}$ maps the terminal nodes to Boolean values.*

The Boolean function f_r represented by a BDD $\mathcal{B} = (N, T, var, low, high, r, val)$ is defined recursively as follows. If $r \in T$ then $f_r = val(r)$; otherwise $f_r = (v \wedge f_{high(r)}) \vee (\neg v \wedge f_{low(r)})$, where $v = var(r)$.

Let $\mathcal{B}_1 = (N_1, T_1, var_1, low_1, high_1, r_2, val_1)$ and $\mathcal{B}_2 = (N_2, T_2, var_2, low_2, high_2, r_2, val_2)$ be two BDDs. We say that $\mathcal{B}_1$ and $\mathcal{B}_2$ are *isomorphic* iff there exists a bijection $\eta\colon N_1 \cup T_1 \to N_2 \cup T_2$ that satisfies the following:

- if $v \in T_1$, then $\eta(v) \in T_2$ and $val_1(v) = val_2(\eta(v))$,
- if $v \in N_1$, then $\eta(v) \in N_2$ with $var(v) = var(\eta(v))$, $\eta(high_1(v)) = high_2(\eta(v))$ and $\eta(low_1(v)) = low_2(\eta(v))$.

We use the following notations when graphically displaying BDDs. Nonterminals are denoted by a circle with the variable labeling the node inside, while terminals are denoted by a rectangle containing the corresponding Boolean value. Solid edges between nodes denote the *high* successors, while dashed edges denote *low* successors.

Let $succ(n) = \{low(n), high(n)\}$, where $n \in N$ is a nonterminal be a relation mapping nonterminals to their successors and let $\mathcal{B} = (N, T, var, low, high, root, val)$ be a BDD. $\mathcal{B}$ is called *ordered* wrt. $\prec$ iff $\forall n \in N (\forall s \in succ(n) : s \in N \implies var(n) \prec var(s))$, that is the variables labeling a nonterminal and its successors are ordered wrt. $\prec$. An example of an Ordered BDD can be seen in Figure 2.1.

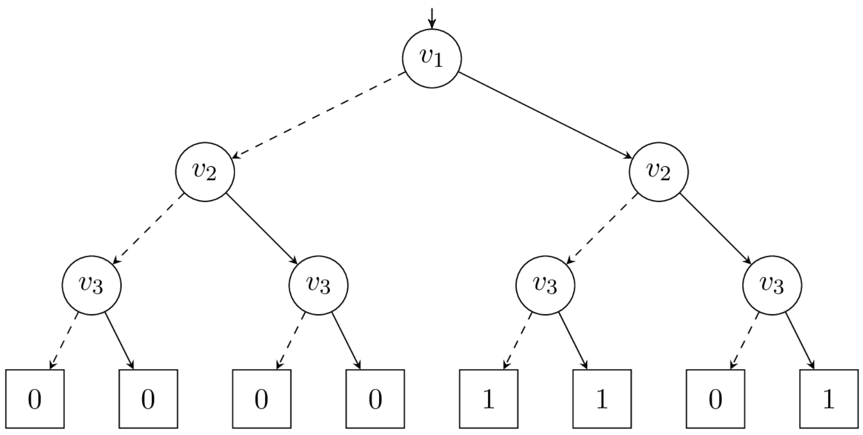


Figure 2.1: Example of an ordered BDD wrt. variable ordering $v_1 \prec v_2 \prec v_3$ encoding the Boolean function $f(v_1, v_{2,v}3) = (v_1 \wedge v_2 \wedge v_3) \wedge (v_1 \wedge \neg v_2)$.

We say that the node $s \in N \cup T$ is *reachable* from the node $p \in N \cup T$ iff there is exists a sequence of nodes $n_1, \dots, n_n$, such that $n_1 = p$, $n_n = s$, and for any n_i, n_{i+1} for $1 \leq i \leq n$ satisfies $n_{i+1} = low(n_i) \vee n_{i+1} = high(n_i)$. We use the predicate *Reachable*(s, p) to denote the reachability of s from p.

Let $f(v_1, \dots, v_n)$ be a Boolean function. A restriction of f with the value of the input variable $v_i \in \{v_1, \dots, v_n\}$ fixed to some value $b \in \mathbb{B}$ is a boolean function $f_{|v_i \leftarrow b} = f(v_1, \dots, v_{i-1}, b, v_{i+1}, \dots, v_n)$. Let $\mathcal{B} = (N, T, var, low, high, r, val)$ be a BDD representing f, and let $\mathcal{S} = low$ if $b = 0$, otherwise $\mathcal{S} = high$. The function $f_{|v_i \leftarrow b}$ is represented by the BDD $\mathcal{B}_{|v_i \leftarrow b} = (N', T', var', low', high', r', val')$, where:

- $N' = N \setminus \{n \mid n \in N \wedge var(n) = v_i\}$,

- $T' = \{t \mid \mathit{Reachable}(t, \mathcal{S}(n)) \wedge var(n) = v_i\}$,
- $low' = (low \setminus \{n \to n' \mid n \to n' \in low \wedge var(n) = v_i\}) \cup \{n \to n'' \mid n \to n' \in low \wedge n' \to n'' \in \mathcal{S} \wedge var(n') = v_i\}$,
- $high' = (high \setminus \{n \to n' \mid n \to n' \in high \wedge var(n) = v_i\}) \cup \{n \to n'' \mid n \to n' \in low \wedge n' \to n'' \in \mathcal{S} \wedge var(n') = v_i\}$,
- $r' = r$ if $var(r) \neq v_i$, otherwise $r' \in \mathcal{S}(r)$,
- $val' = \{t \to b \mid t \to b \in val \wedge t \in T'\}$.

Let $\mathcal{B}$ be a BDD. We define *subgraph BDD* rooted by $v \in N \cup T$ to be a BDD $\mathcal{B}_1 = (N', T', var', low', high', v, val')$ where $N' \subseteq N$ is the set of nonterminals reachable from v, $T' \subseteq T$ is the set of terminals reachable from v, $var' = var|_{N'}$, $high' = high|_{N'}$, $low' = low|_{N'}$, $val' = low|_{T'}$.

Definition 2.3.2. *A BDD is called* reduced *if there is no nonterminal node $n \in N$, such that $low(n) = high(n)$, and it does not contain any nodes $v_1, v_2 \in N \cup T, v_1 \neq v_2$, such that the subgraph BDDs rooted by those nodes are isomorphic.*

A BDD that is ordered and reduced is called Reduced Ordered BDD (ROBDD). The combination of these restrictions on the BDD structure allows ROBDDs to compactly represent Boolean functions while having a canonical form. Given the existence of algorithms efficiently manipulating ROBDDs, they are the most frequently used BDD variant, to the point that they are being referenced simply as BDDs. Figure 2.2 provides an example of an ROBDD.

Boolean functions represented by BDDs can be combined using the *Apply* procedure (Algorithm 3). The procedure computes a BDD $\mathcal{B}_{f \diamond g}$ representing a function $f \diamond g$ from BDDs $\mathcal{B}_f$ and $\mathcal{B}_g$ representing corresponding functions f and g, where $\diamond \colon \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ is some Boolean operator. Algorithm 3 uses a table to cache the computed results of applying $\diamond$ to subgraph BDDs to increase it efficiency.

Algorithm 3 Construction of BDD $\mathcal{B}_{f \diamond g}$

```
Input: ROBDD $\mathcal{B}^f = (N_f, T_f, var_f, low_f, high_f, r_f, val_f)$ wrt. $\prec$ representing $f\colon \mathbb{B}^n \to \mathbb{B}$,
    ROBDD $\mathcal{B}^g = (N_g, T_g, var_g, low_g, high_g, r_g, val_g)$ wrt. $\prec$ representing $g\colon \mathbb{B}^n \to \mathbb{B}$,
    Boolean operator $\diamond : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$
Output: ROBDD $\mathcal{B}^{f \diamond g} = (N_f, T_f, var_f, low_f, high_f, r_f, val_f)$ representing $f \diamond g$
 1: function APPLY($\mathcal{B}^f, \mathcal{B}^g, \diamond$)
 2:     if $r_f \in T_f$ and $r_g \in T_g$ then
 3:         let $t$ be a new terminal node
 4:         $val_t \leftarrow val_f(r_f) \diamond val_g(r_g)$
 5:         $\mathcal{B}' \leftarrow (\emptyset, \{t\}, \emptyset, \emptyset, \emptyset, t, \{t \to val_t\})$
 6:         return $\mathcal{B}'$
 7:     end if
 8:     if $(\diamond, root_f, root_g)$ are in table then
 9:         return table[$(\diamond, root_f, root_g)$]
10:     end if
11:     $v \leftarrow var(r_g)$ if $var(r_g) \prec var(r_f)$ else $var(r_f)$
12:     $\mathcal{B}_l = (N_l, T_l, var_l, low_l, high_l, r_l, val_l) \leftarrow Apply(\mathcal{B}^f_{|v \leftarrow 0}, \mathcal{B}^g_{|v \leftarrow 0}, \diamond)$
```

```
13:     $\mathcal{B}_h = (N_h, T_h, var_h, low_h, high_h, r_h, val_h) \leftarrow Apply(\mathcal{B}^f_{|v\leftarrow 1}, \mathcal{B}^g_{|v\leftarrow 1}, \diamond)$
14:     if $Isomorphic(\mathcal{B}_l, \mathcal{B}_h)$ then
15:         return $\mathcal{B}_h$
16:     end if
17:     let $n$ be a new nonterminal node
18:     $low' \leftarrow low_l \cup low_h \cup \{n \rightarrow r_l\}$
19:     $high' \leftarrow high_l \cup high_h \cup \{n \rightarrow r_h\}$
20:     $var' \leftarrow var_l \cup var_h \cup \{n \rightarrow v\}$,
21:     $\mathcal{B}' \leftarrow (N_l \cup N_h \cup \{n\}, T_l \cup T_h, var', low', high', n, val_l \cup val_h)$
22:     add $(\diamond, root_f, root_g) \rightarrow \mathcal{B}'$ to table
23:     return $\mathcal{B}'$
24: end function
```

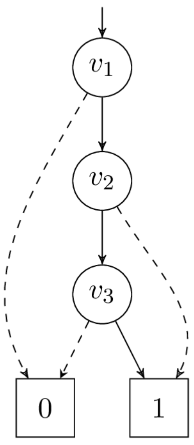


Figure 2.2: Example of a reduced ordered BDD wrt. the variable ordering $v_1 \prec v_2 \prec v_3$ encoding the Boolean function $f(v_1, v_2, v_3) = (v_1 \wedge v_2 \wedge v_3) \vee (v_1 \wedge \neg v_2)$.

Multi-terminal binary decision diagrams (MTBDDs) are an extension of the classical BDDs, based on the observation that there are no obstacles preventing the terminal nodes to have their values limited only to two values. By relaxing this restriction and allowing the terminal values to be any value from a non-empty set S, they present a compact representation of any function $f\colon \mathbb{B}^n \rightarrow S$. Therefore, a MTBDD $\mathcal{M}$ is a 5-tuple $(N, T, var, low, high, root, val)$ with the same semantics as BDDs have, with the only difference being $val\colon T \rightarrow S$. Ordered MTBDDs are defined in the same fashion as BDDs, same as their reduced variant, as the definition of isomorphic BDDs can be easily extended to MTBDDs. An example of an MTBDD can be seen in Figure 2.3. Similarly, Algorithm 3 can be extended to MTBDDs in a straightforward fashion. Let $f\colon \mathbb{B}^n \rightarrow S_f$ and $g\colon \mathbb{B}^n \rightarrow S_g$ be two functions represented by the corresponding MTBDDs $\mathcal{M}_f$ and $\mathcal{M}_g$, and let $\diamond\colon S_f \times S_g \rightarrow S$ for some nonempty set S. We write $\mathcal{M}_f \diamond \mathcal{M}_g$ to denote the application of the function $\diamond$ to the MTBDDs $\mathcal{M}_f$ and $\mathcal{M}_g$ using Algorithm 3 extended to MTBDDs.

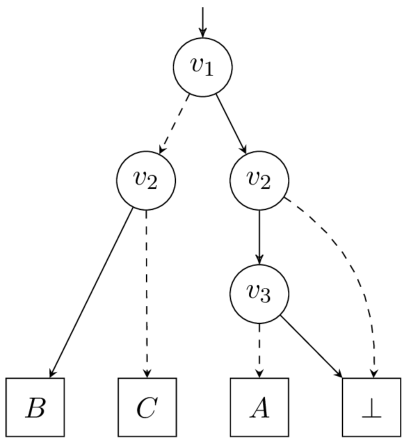


Figure 2.3: An MTBDD of the partial function $f(v_1, v_2, v_3) = \{((1,1,0), A), ((0,1,?), B), ((0,0,?), C)\}$ where the ? symbol stands for a don't-care bit (the value of the function does not depend on it). Terminal node with value $\perp$ denotes that the function is undefined for the input leading to this node.

Chapter 3

Automata-based decision procedure for Presburger arithmetic

The idea of a decision procedure that utilizes finite automata to encode and manipulate the solution space of a given formula and its fragments was first given by Büchi in 1960 [7]. In this work Büchi first introduced the notion of formulae being represented via finite automata and employed this observation to develop an algorithm deducing the formula satisfiability using this formalism. However, the automata-based approach to determining whether a formula has a model was not applied to PrA until 1996 when Boudet & Comon [5] published their work on constructing compact automata encoding the solution space of atomic constraints.

The idea of representing PrA formulae as automata is based on the fact that, given some base $b \geq 2$, it is possible to encode any natural number $n \in \mathbb{N}$ as a word $w = d_k \cdots d_0$, where $\forall i(0 \leq d_i < b - 1)$, so that $n = \sum_{i=0}^{k} b^i d_i$ holds. The integer b is called the base, while d_i are the digits of the number. This encoding can be further extended to include all integers by requiring that all encodings of an integer have a minimum of $p \geq 2$ digits. Let z be an integer such that $-b^{p-1} \leq z < b^{p-1}$, where p is the number minimal number of digits used in a given encoding. If $z < 0$ then z will be encoded as the last p digits of $b^m + z$, $\forall m > p - 1$. According to such a scheme, the leftmost digit of the encoding of z can be thought of as the one determining the sign of the encoded number — if z is negative, then the leftmost digit is $b - 1$, otherwise the leftmost digit is 0. It is worth pointing out that one number has infinitely many encodings in the scheme described above which can be obtained by repeating the sign digit.

Let φ be a PrA formula with n free variables, and let $Sol(\varphi)$ denote the solution space of φ. Then φ is represented via an automaton $\mathcal{A}_\varphi$ iff the language of the automaton $\mathcal{L}(\mathcal{A}_\varphi)$ precisely contains only $Sol(\varphi)$ encoded using some encoding.

3.1 *LSBF* encoding

In order to encode models of Presburger formulae we use the binary Least Significant Bit First (LSBF) encoding. Therefore, the alphabet used by automata throughout the decision procedure is $\Sigma = \mathbb{B}^n$ where n is the number of free variables in the given formula. To provide an illustration, consider the following tuple of integers:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} -3 \\ 10 \end{pmatrix}$$

Using the standard two's complement (the most significant bit is at the first position), the tuple is encoded as follows:

$$\begin{aligned} -3_{10} &= 101_2 \\ 10_{10} &= 01010_2 \end{aligned}$$

The first bit from the left is always the sign bit, however, different numbers might have a different length of their representation. As each of these encoded numbers will be written on a separate track on automaton's input tape, it is required that they all share the same length. This can be achieved by adding padding i.e. repeating the sign bit, which does not change the represented value. In our example, the result would be the following:

$$\begin{aligned} -3_{10} &= 11101_2 \\ 10_{10} &= 01010_2 \end{aligned}$$

Finally, the binary representations can be broken into bit tuples containing the i-th bit of each of the encoded number. Since the encoding is least significant bit first, the order of tuples is reversed.

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} -3 \\ 10 \end{pmatrix} = \left(\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right)_{LSBF}$$

As a number represented in two's complement might have infinitely many encodings obtained by repeating the sign bit and the constructed automaton must accept every such an encoding, should this be an encoded solution to the formula represented by the constructed automaton. We will refer to this requirement as the *saturation property*.

The choice of the LSBF encoding was made as it is commonly used in context of this decision procedure. However, there is some literature dealing with the binary Most Significant Bit First (MSBF) encoding, e.g., [20].

3.2 Constructing automata from atomic constrains

In the following section we introduce the algorithms used to construct automata encoding the solution space of atomic constraints. The presented algorithms can be seen as a bridge between the realm of first-order logic to the realm of finite automata. As we rely on the LSBF encoding, the presented constructions construct automata with alphabet $\Sigma = \mathbb{B}^n$ where n is the number of variables in a given formula. We provide constructions for atomic formulae with solutions in $\mathbb{N}$, and also for formulae that have their solution space in $\mathbb{Z}$.

3.2.1 Inequations

Let $\varphi_{\leq} : \vec{a} \cdot \vec{x} \leq c$ be an inequality, where $\vec{x}$ is a vector of variables of size n, $\vec{a} \in \mathbb{N}^n$ denotes the vector of variable coefficients, and $c \in \mathbb{Z}$ is a constant.

The procedure *IneqToDFA* in Algorithm 4 (adoped from [14]) constructs an automaton $A_{\varphi_\leq}$ encoding $Sol(\varphi_\leq)$ using the LSBF encoding. An example of the constructed automaton for $\varphi\colon x \leq 4$ over $\mathbb{N}$ can be seen in Figure 3.1.

Algorithm 4 Construction of an DFA encoding solutions of an inequality $\varphi_\leq$ over $\mathbb{N}$

```
Input: An inequality $\varphi_\leq\colon \vec{a} \cdot \vec{x} \leq b$ over $\mathbb{N}$
Output: DFA $A_{\varphi_\leq} = (Q, \mathbb{B}^{|\vec{x}|}, \delta, Q_0, F)$ that encodes $\varphi_\leq$
 1: function IneqToDFA($\varphi_\leq$)
 2:     $Q, \delta, F \leftarrow \emptyset$
 3:     $Q_0 \leftarrow \{q_b\}$
 4:     $W \leftarrow \{q_b\}$
 5:     while $W \neq \emptyset$ do
 6:         $s_k \leftarrow$ pick and remove state from $W$
 7:         add $s_k$ to $Q$
 8:         for every $\sigma \in \Sigma$ do
 9:             $v \leftarrow \lfloor \frac{1}{2}(k - \vec{a} \cdot \sigma) \rfloor$
10:             if $q_v \notin Q$ then
11:                 add $q_v$ to $W$
12:             end if
13:             add $(q_k, \sigma, q_v)$ to $\delta$
14:             if $v \geq 0$ then
15:                 add $q_v$ to $F$
16:             end if
17:         end for
18:     end while
19:     return $(Q, \Sigma, \delta, Q_0, F)$
20: end function
```

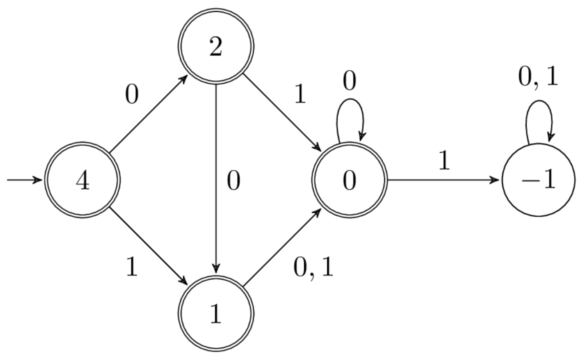


Figure 3.1: Automaton A_φ for the inequality $\varphi\colon x \leq 4$ over $\mathbb{N}$

Sacrificing the determinism, it is also possible to construct an automaton that encodes the solutions of the same inequation φ over integers, utilizing the two's complement encoding [14]. The nondeterminism is introduced due to the used encoding — the sign of the number can be determined only after reading the last symbol on the input tape. This is reflected in the modified procedure *IneqToNFA* in Algorithm 5 at lines 14 and 15, where a scenario in which the current symbol σ is the last one is considered. Therefore, the symbol is interpreted according to the used two's complement encoding as a vector of sign bits and it is checked whether such input would satisfy the inequality φ. An example of an automaton encoding the solution space of $\varphi : x \leq 4$ over $\mathbb{Z}$ is depicted in Figure 3.2.

Algorithm 5 Construction of an NFA encoding solutions of an inequality $\varphi_{\leq}$ over $\mathbb{Z}$

Input: An inequality $\varphi_{\leq}\vec{a} \cdot \vec{x} \leq b$ over $\mathbb{Z}$
Output: NFA $A_{\varphi_{\leq}} = (Q, \Sigma, \delta, Q_0, F)$ that encodes $\varphi_{\leq}$

```
 1: function IneqToNFA($\varphi_{\leq}$)
 2:     $Q, \delta, F \leftarrow \emptyset$
 3:     $Q_0 \leftarrow \{q_b\}$
 4:     $W \leftarrow \{q_b\}$
 5:     while $W \neq \emptyset$ do
 6:         $s_k \leftarrow$ pick and remove state from $W$
 7:         add $s_k$ to $Q$
 8:         for every $\sigma \in \Sigma$ do
 9:             $v \leftarrow \lfloor \frac{1}{2}(k - \vec{a} \cdot \sigma) \rfloor$
10:             if $q_v \notin Q$ then
11:                 add $q_v$ to $W$
12:             end if
13:             add the transition $(q_k, \sigma, q_v)$ to $\delta$
14:             $v' \leftarrow \frac{1}{2}(k + \vec{a} \cdot \sigma)$
15:             if $v' \geq 0$ then
16:                 if $F = \emptyset$ then
17:                     add $q_f$ to $Q$ and $F$
18:                 end if
19:                 add the transition $(q_k, \sigma, q_f)$ to $\delta$
20:             end if
21:         end for
22:     end while
23:     return $(Q, \Sigma, \delta, Q_0, F)$
24: end function
```

3.2.2 Equations

Let $\varphi_{=}$: $\vec{a} \cdot \vec{x} = c$ be an equality, where $\vec{x}$ is a vector of variables of size n, $\vec{a} \in \mathbb{N}^n$ denotes the vector of variable coefficients, and $c \in \mathbb{Z}$ is a constant.

Similarly as with inequations, a direct construction for an automaton $A_{\varphi_{=}}$ encoding the solutions to $\varphi_{=}$ exists [14]. The *EqToDFA* procedure resembles the construction for automata encoding solution space of inequations, however, it does not rely on the floor operation when calculating the label of the transition destination state. Should the difference of the number labeling the currently processed state and the symbol on the input tape (weighted by the corresponding coefficients) be odd, the number encoded on the input tape and the absolute part of the equation must differ in some bit, and therefore, they are cannot be equal.

3.2.3 Congruences

Let $\varphi_{\equiv}$: $\vec{a} \cdot \vec{x} \equiv_m c$ be a PrA formula, where $\vec{x}$ is a vector of variables, $\vec{x}$ is a vector of variable coefficients, $m \in \mathbb{N}$ a constant called divisor, and $c \in \mathbb{N}$ a constant. The solution space of $\varphi_{\equiv}$ can be encoded using Algorithm 6 (adopted from [12]).

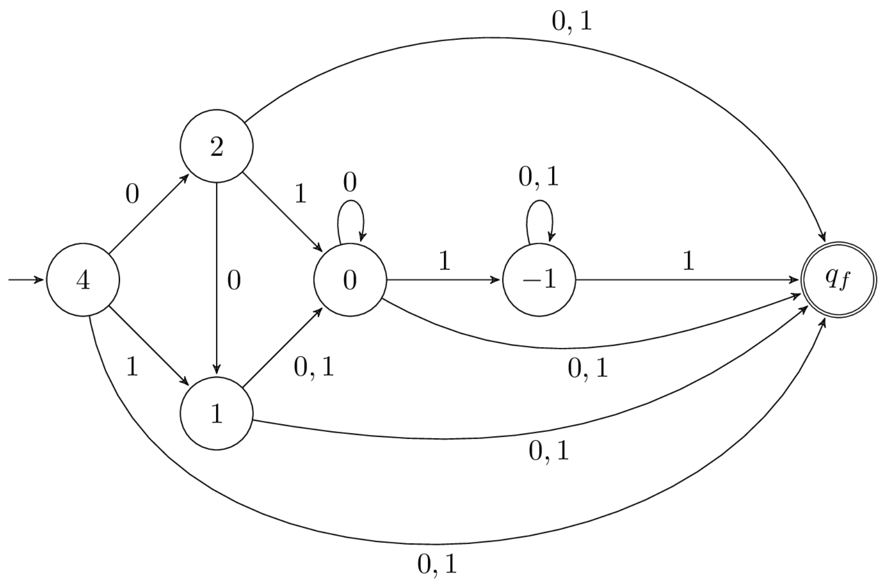


Figure 3.2: Automaton A_φ for the inequality $\varphi : x \leq 4$ over $\mathbb{Z}$

Algorithm 6 Construction of an NFA encoding solutions of a congruence $\varphi_\equiv$ over $\mathbb{Z}$

```
Input: A congruence $\varphi_\equiv : \vec{a} \cdot \vec{x} \equiv_m c$
Output: NFA $A_{\varphi_\equiv} = (Q, \Sigma, \delta, Q_0, F)$ that encodes $\varphi_\equiv$
 1: function CongruenceToNFA($\varphi$)
 2:     $Q, \delta \leftarrow \emptyset$
 3:     $F \leftarrow \{q_f\}$
 4:     $Q_0 \leftarrow \{q_{\vec{a}\cdot\vec{x}\equiv_m c}\}$
 5:     $W \leftarrow \{q_{\vec{a}\cdot\vec{x}\equiv_m c}\}$
 6:     while $W \neq \emptyset$ do
 7:         $q_{\vec{a}\cdot\vec{x}\equiv_n d} \leftarrow$ pick and remove state from $W$
 8:         add $q_{\vec{a}\cdot\vec{x}\equiv_n d}$ to $Q$
 9:         for every $\sigma \in \Sigma$ do
10:             if $2 \mid n$ then and $2|(d - \vec{a} \cdot \vec{\sigma})$
11:                 $n' \leftarrow n/2$
12:                 $d' \leftarrow (d - \vec{a} \cdot \vec{\sigma})/2$
13:                 add $(q_{\vec{a}\cdot\vec{x}\equiv_n d}, \sigma, q_{\vec{a}\cdot\vec{x}\equiv_{n'} d'})$ to $\delta$
14:             else if $2 \nmid n$ then
15:                 if $2|(d - \vec{a} \cdot \vec{\sigma})$ then
16:                     $d' \leftarrow (d - \vec{a} \cdot \vec{\sigma})/2$
17:                 else
18:                     $d' \leftarrow (d + n - \vec{a} \cdot \vec{\sigma})/2$
19:                 end if
20:                 add $(q_{\vec{a}\cdot\vec{x}\equiv_n d}, \sigma, q_{\vec{a}\cdot\vec{x}\equiv_n d'})$ to $\delta$
21:                 if $q_{\vec{a}\cdot\vec{x}\equiv_n d'} \notin W$ then
22:                     add $q_{\vec{a}\cdot\vec{x}\equiv_n d'}$ to $W$
23:                 end if
```

```
24:             if $d + \vec{a} \cdot \vec{\sigma} \equiv_n 0$ then
25:                 add $(q_{\vec{a}\cdot\vec{x}\equiv_n d}, \sigma, q_f)$ to $\delta$
26:             end if
27:         end if
28:     end for
29: end while
30: return $(Q, \Sigma, \delta, Q_0, F)$
31: end function
```

3.3 Automata-based decision procedure for Presbuger arithmetic

The automata-based decision procedure works in a bottom-up fashion, starting with the atomic PrA subformulae found in the input formula φ and working its way up. Using the algorithms *IneqToNFA*, *EqToNFA*, and *CongruenceToNFA*, it constructs the automata for the atomic PrA subformulae found in φ. These automata are then modified and combined according to the structure of φ, mapping logical conjunctions to their equivalent operations on the automata and languages they represent:

- *Negation* $\neg\psi(x_1, x_2, ..., x_n) \rightsquigarrow A^C_\psi$, where A^C_ψ encodes $L(\overline{A_\psi})$ (Automaton complement),
- *Conjunction* $\varphi(x_1, x_2, ..., x_n) \wedge \psi(x_1, x_2, ..., x_n) \rightsquigarrow L(A_\varphi) \cap L(A_\psi)$, and
- *Disjunction* $\varphi(x_1, x_2, ..., x_n) \vee \psi(x_1, x_2, ..., x_n) \rightsquigarrow L(A_\varphi) \cup L(A_\psi)$.

The *existential quantification* $\exists x(\psi)$ is performed via projecting away the track corresponding to the variable x from the input tape. There is no equivalent operation for the *universal quantification*, and therefore, the quantifier is replaced by an existential one according to the law: $\forall x(\psi) \equiv \neg\exists x(\neg\psi)$.

$$\begin{tikzcd}
\widetilde{A_\varphi} \arrow[r] & q_{-2} \arrow[loop above, "\begin{smallmatrix} x \\ y \end{smallmatrix}\begin{bmatrix} \cancel{0} \\ 1 \end{bmatrix}"] \arrow[r, "\begin{smallmatrix} x \\ y \end{smallmatrix}\begin{bmatrix} \cancel{1} \\ 0 \end{bmatrix}{,}\begin{bmatrix} \cancel{0} \\ 0 \end{bmatrix}{,}\begin{bmatrix} \cancel{1} \\ 1 \end{bmatrix}"] \arrow[rrd, bend right, "\begin{smallmatrix} x \\ y \end{smallmatrix}\begin{bmatrix} \cancel{?} \\ 1 \end{bmatrix}"] & q_{-1} \arrow[loop above, "\begin{smallmatrix} x \\ y \end{smallmatrix}\begin{bmatrix} \cancel{0} \\ 1 \end{bmatrix}{,}\begin{bmatrix} \cancel{0} \\ 0 \end{bmatrix}{,}\begin{bmatrix} \cancel{1} \\ 1 \end{bmatrix}"] \arrow[r, "\begin{smallmatrix} x \\ y \end{smallmatrix}\begin{bmatrix} \cancel{0} \\ 1 \end{bmatrix}"] & q_f \\
 & & & q'_f
\end{tikzcd}$$

Figure 3.3: An illustration of a missing model encoding on a fragment of automaton for $\varphi : \exists x(x - y \geq 12)$. The rest of the automaton is denoted as $\widetilde{A_\varphi}$. The transition and the final state added by the *PadClosure* algorithm are displayed in teal.

After a variable track is projected away, the language of the resulting automaton might not necessarily contain all encodings of the solutions. Removing a variable track naturally truncates the alphabet, and therefore, some transitions along symbols that were not previously a part of the padding of an encoded solution might become a part of the padding.

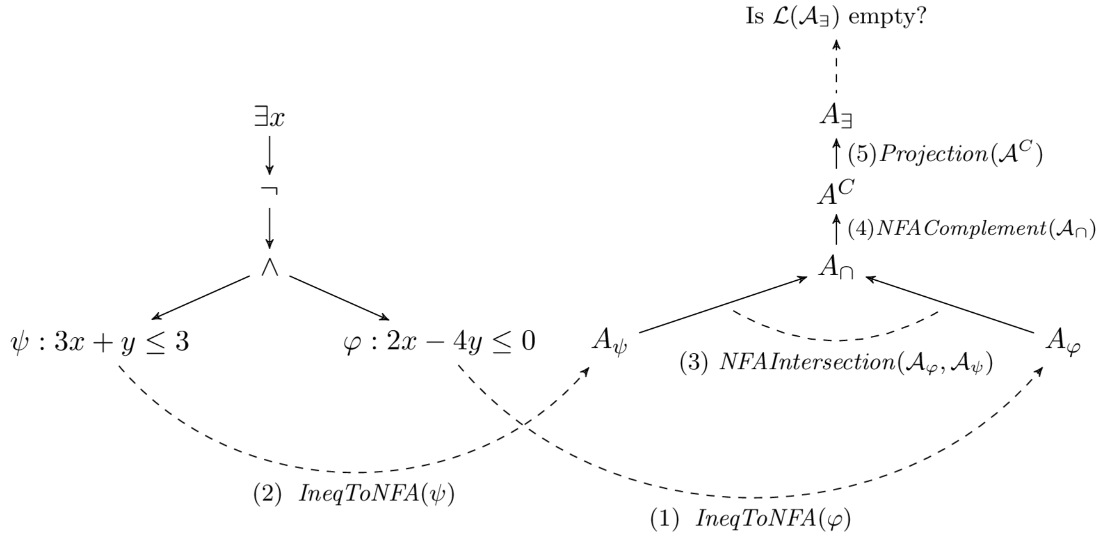
This situation is illustrated in Figure 3.3, which depicts a problematic fragment of automaton for the formula $\varphi : \exists x(x - y \geq 12)$. This inconsistency can be solved by using the *PadClosure* procedure in Algorithm 7 that augments the automaton structure so that it satisfies the saturation property.

Algorithm 7 Augment $\mathcal{A}$ so that it accepts all encodings of models.

```
Input: NFAs $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$
Output: NFA $\mathcal{A}' = (Q', \Sigma, \delta', Q_0, F')$ accepting all encodings of models
 1: function $PadClosure(\mathcal{A})$
 2:     $\delta' \leftarrow \delta$, $W \leftarrow \emptyset$
 3:     $q_f \leftarrow$ a new state to be added to $\mathcal{A}$ such that $q_f \notin Q$
 4:     for $\zeta \in \Sigma$ do
 5:         $S \leftarrow \emptyset$, $W \leftarrow Pre_\delta(F)$
 6:         while $W \neq \emptyset$ do
 7:             $q \leftarrow$ pick and remove from $W$
 8:             add $s$ to $S$
 9:             for $q' \in Pre_\delta(q)$ do
10:                 add $q'$ to $W$ if $q' \notin S$
11:             end for
12:         end while
13:         for $q \in S$ do
14:             add $(q, \zeta, q_f)$ to $\delta'$ if $Post_\delta(q, \zeta) \cap F = \emptyset$
15:         end for
16:     end for
17:     return $(Q, \Sigma, \delta, Q_0, F)$ if $\delta = \delta'$ else $(Q \cup \{q_f\}, \Sigma, \delta', Q_0, F \cup \{q_f\})$
18: end function
```

Algorithm 7 works as follows. For every possible padding symbol $\zeta \in \Sigma$ the set S of states accepting a word ζ^+ consisting only of the padding symbol. Therefore, the set S contains all states for which the saturation property might be broken. The property is trivially broken for a state $s \in S$ if the word $w = \zeta$ is not accepted from s, since it follows from the way S is constructed that there must be a word $w' = \zeta \ldots \zeta$, $w' \in \mathcal{L}(s)$ consisting only the padding symbol. For any word w' with $|w'| \geq 2$, there exists a run of the input automaton $s \overset{u}{\Longrightarrow} s'$ over a word u, such that $s' \in S$ and $w' = u\zeta$. Therefore, it is sufficient to only check whether the property is trivially broken in any state $s \in S$, as all cases of saturation property being broken by w' will be considered by checking whether the property is trivially broken in every $s' \in S$.

After the entire input formula ψ is processed in the fashion described above, we are left with an automaton $\mathcal{A}_\psi$ encoding the solution space of ψ. Therefore, the existence of a model can be determined by checking whether the $\mathcal{L}(\mathcal{A}_\psi)$ is empty, which would signify that ψ has no solution. The emptiness checking of $\mathcal{L}(\mathcal{A}_\psi)$ can be performed by standard graph searching algorithms such as Depth Fist Search (DFS). An illustration of the entire decision procedure can be found in Figure 3.4.



Is $\mathcal{L}(\mathcal{A}_\exists)$ empty?
$\exists x$
$\neg$
$\wedge$
$\psi : 3x + y \leq 3$
$\varphi : 2x - 4y \leq 0$
$A_\exists$
(5) $\mathit{Projection}(\mathcal{A}^C)$
A^C
(4) $\mathit{NFAComplement}(\mathcal{A}_\cap)$
$A_\cap$
A_ψ
A_φ
(3) $\mathit{NFAIntersection}(\mathcal{A}_\varphi, \mathcal{A}_\psi)$
(2) $\mathit{IneqToNFA}(\psi)$
(1) $\mathit{IneqToNFA}(\varphi)$

Figure 3.4: Illustration of the decision procedure for the input formula $\alpha\colon \exists x(\neg(3x + y \leq 3 \wedge 2x - 4y \leq 0)$ depicting how the input formula structure is mapped onto operations on regular languages, and the order of the performed transformations. Atomic PrA formulae are denoted by φ and ψ.

Chapter 4

Description of the implemented automata-based SMT solver

This chapter describes *Amaya* - an experimental SMT solver based on finite automata implemented as a part of this work. The chapter starts by providing an overview of Amaya's architecture, and the steps our solver performs in order to decide whether the input formula has a model. The chapter then continues with a detailed description of the developed symbolic representation of transitions using MTBDDs, as well as the developed MTBDD-aware versions of the classical automata constructions. The chapter concludes with a short description of Amaya's additional features, supporting experimentation with the automata-based decision procedure.

4.1 Architecture of the implemented solver

Amaya is implemented in the Python 3 programming language. The choice of the implementation language was driven by the goal of this work — to create an experimental, automata-based SMT solver for PrA, laying the necessary foundation for the future research of the automata-based decision procedure and its practical applications. Python presented itself as a great choice, due to its dynamic nature and its vast library ecosystem, providing more flexibility for experimentation compared to compiled statically typed languages. Naturally, this choice greatly influences runtime performance, but besting the state-of-the-art SMT solvers in terms of speed is not the topmost priority of the created tool.

Our solver was designed to support SMT-LIB [4] — a standardized input language supported by the state-of-the-art solvers, and thus allowing easy performance comparisons of our solver to the solvers implementing other decision procedures. SMT-LIB syntactically belongs to the family of Lisp-like languages, implying that the lexical analysis and the syntactical analysis are easy. The internal representation produced by the syntactical analysis — the abstract syntax tree (AST) — is based on Python's built-in lists. This representation is used as it closely matches the structure of the input text, and it is easy to implement, allowing us to focus on the actual decision procedure. Furthermore, the internal representation based on lists has proven to be very flexible, especially when performing preprocessing.

We carefully picked the features of SMT-LIB to support in our implementation based on their occurrence in available benchmarks. Except the features absolutely necessary to encode input formulae, we also support `let` expressions allowing binding a value to a vari-

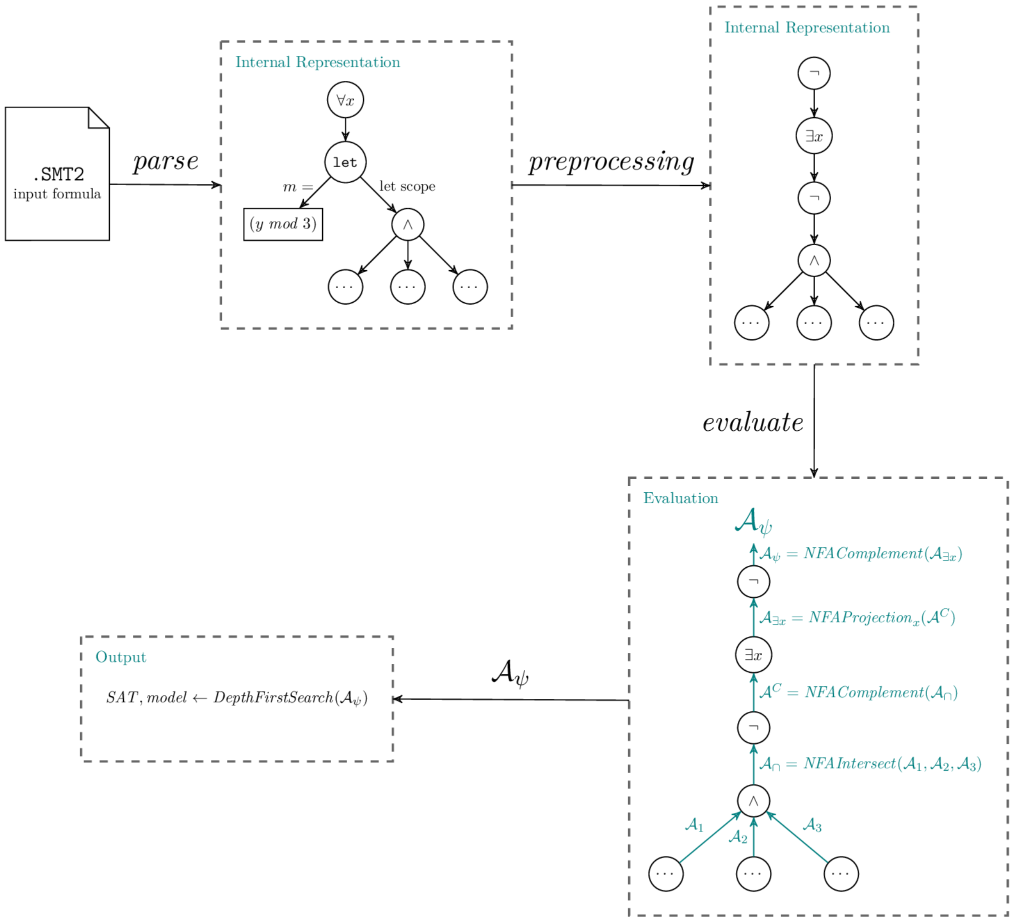

.SMT2
input formula
parse
Internal Representation
$\forall x$
let
$m =$
let scope
$(y \ mod \ 3)$
$\wedge$
preprocessing
Internal Representation
$\neg$
$\exists x$
$\neg$
$\wedge$
evaluate
Evaluation
$\mathcal{A}_\psi$
$\mathcal{A}_\psi = NFAComplement(\mathcal{A}_{\exists x})$
$\mathcal{A}_{\exists x} = NFAProjection_x(\mathcal{A}^C)$
$\mathcal{A}^C = NFAComplement(\mathcal{A}_\cap)$
$\mathcal{A}_\cap = NFAIntersect(\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3)$
$\mathcal{A}_1$
$\mathcal{A}_2$
$\mathcal{A}_3$
$\mathcal{A}_\psi$
Output
$SAT, model \leftarrow DepthFirstSearch(\mathcal{A}_\psi)$

Figure 4.1: Diagram illustrating the operations Amaya has to perform in order to deduce the SAT value from the given input SMT2 text.

able in a certain scope and `ite` (if-then-else) expressions. Both of these constructs are in essence a syntax sugar providing compact notation for input formulae. In order to support this syntax sugar, together with some trivial optimizations on the input formulae, the solver performs multiple preprocessing passes. The number of preprocessing AST traversals is lower than the number of overall syntactical modification performed, as some AST modifications do not conflict (modify the same AST nodes) nor depend on other preprocessing changes. Such non-conflicting AST modifications can be combined into a single pass, avoiding needless AST traversals. The preprocessing algorithm is implemented in an abstract fashion, allowing extending a preprocessing pass just by defining which function should be called if a given node is encountered when traversing the AST. The preprocessing is performed in three passes:

1. The first pass performs the macro expansion of the `let` expressions by replacing all the occurrences of the variables bound in a let expression by the their value in the scope of the expression. After all bound variables have been folded in, the `let` expression is removed from the AST. However, if a bound value can be represented as an automaton (e.g. its value is an entire formula), variable occurrences are not replaced by its value, but the variable value is looked up during the evaluation when the variable is referenced. The automaton created for such a formula is cached and

subsequent variable occurrences use the cached value without building the automaton again.

2. During the second pass, universal quantifiers are replaced by existential quantifiers using the law $\forall x(\eta) \Leftrightarrow \neg\exists x(\neg\eta)$. Similarly, as there is no equivalent automaton operation for implication, implications must also be replaced using the law: $\varphi \rightarrow \psi \Leftrightarrow (\neg\varphi) \vee \psi$. Finally, `ite` expressions are also removed into expression they abbreviate: `(ite b x y)⇔(or (and b x) (and (not b) y))`.

3. The last pass removes double negations according to the law: $\neg(\neg\psi) \Leftrightarrow \psi$.

The core of the implemented SMT solver is an automata library providing data structures for storing automata and implementing automata procedures. The provided procedures range from algorithms crucial for the decision procedure (e.g. intersection construction) to algorithms serving for experimentation such as various minimization procedures. The automata library represents automata by a separate class, originally designed in a fashion matching the formal definition of an automaton. However some restrictions had to be put forth in order to efficiently utilize MTBDDs to speed up automata operations (MTBDDs are discussed bellow). Automata transition relations are abstracted into a separate class, due to the need to experiment with different ways of representing transitions. The need to be able to seamlessly change between the different ways of storing the transition function is due to its big impact on the overall decision procedure runtime.

As quantifiers facilitate the possibility of a variable name being ambiguous, the solver has to track variable scopes throughout the decision procedure. As a consequence, variables are no longer identified by their name, but rather a unique identifier assigned to them based on the context they are in. Similar context analysis is required by the variables introduced in the `let` expression. Naturally, the scopes introduced by `let` expressions and quantifiers are semantically different, and therefore, the solver must keep two separate stacks of scopes.

Alphabet symbols are not stored explicitly, due to the exponential growth of the alphabet size wrt. the number of variables in the input formula resulting in high memory consumption. Instead, the class encapsulating alphabet stores the variable identifiers and their names, and generates the alphabet symbols on demand. Initially, every automaton had a separate, potentially different alphabet based on the variables of the formula the automaton was encoding. However, adding support for MTBDDs required to have one unified alphabet due to the used MTBDD library requiring pre-declaring all variables. The negative impact on the performance of the classical automata algorithms iterating over the entire alphabet is mitigated by explicitly tracking what variables are used in the formula encoded by the automaton, allowing algorithms to iterate only over symbols differing in significant tracks. Amaya supports a simple symbolic representation of alphabet symbols by providing a notation for a *don't-care* bit. If a transition $q \xrightarrow{s} q'$ has the value of the i-th bit of s set as don't-care, the transition represents two transitions — one with the i-th bit set to zero, and the other one with the i-th bit set to one. This symbolic representation reduces required memory used by the automaton's transition relation. The automata library does not actively perform detection and compression of don't-care bits. However, transition symbols are compressed when exporting automata, as the exported automata are often visually inspected by a human, and the compressed symbols greatly help readability.

After preprocessing is performed, all AST subtrees representing atomic relations are converted into leaves containing the atomic relations in a normalized from which automata can be created directly using the constructions described in Section 3.2. In this form every variable occurs only once on the left-hand side of the predicate symbol and on the right-hand side is a constant, therefore, arithmetical expressions inside the relation must be evaluated and the relation rearranged. As Amaya supports also modulo terms, the performed normalization must also compute the coefficients of these constraints similarly to the variable coefficients. The conversion of AST subtrees representing atomic relations is performed as a standalone step in order to separate the decision procedure from the normalization code manipulating arithmetic expressions.

As every SMT-LIB theory declaration contains implicitly the `Core` theory providing the definition of the `Bool` sort (type) and the usual set of Boolean connectives, the input formulae can contain the equality symbol used to denote PrA atomic constraints, or to denote equivalence of two formulae. Our solver has to distinguish between these scenarios, as a new automaton must be constructed for atomic constraints, whereas the equivalence of two formulae operates on two automata representing the corresponding formulae. The ambiguity of equality cannot be addressed in preprocessing, as it is not possible to determine whether the expression `(= x y)` is a PrA atomic constraint without having contextual information about the sorts of x and y.

$$
\begin{aligned}
&\vec{a} \cdot x + (y \; mod \; M) \leq c \Leftrightarrow \\
&\exists m(\vec{a} \cdot x + m \leq c \land 0 \leq m \leq M - 1 \land m \equiv_M y) \qquad (4.1)\\
&\exists k(\vec{a} \cdot x + (y - kM) \leq c \land 0 \leq (y - kM) \leq M - 1) \qquad (4.2)
\end{aligned}
$$

Supporting modulo terms presents a problem when constructing automata for atomic constraints. The solver must detect that a normalized constraint contains modulo terms, and rewrite the constraint using existential quantifiers as shown in Formula 4.1 in order to express the modulo terms since there is no construction for atomic constrains containing modulo terms. The solver could rewrite the modulo constraints in an alternative fashion as shown in Formula 4.2, however, our implementation prefers the earlier formula, as the Algorithm 6 constructs automata of a more interesting structure (discussed in Section 6.1). Furthermore, there is no dedicated function in the SMT-LIB LIA (linear integer arithmetic) theory for expressing congruences, therefore, congruences are written as an equality of a modulo term and a constant. For example, the congruence $x \equiv_3 1$ would be written as `(mod x 3) = 1`. Such syntactical structures must be correctly recognized, and not rewritten using an existential quantifier.

After all relation subtrees are converted to directly evaluable leaves, the solver traverses the AST constructing and combining intermediate automata as described in Section 3.3, utilizing the algorithms provided by the core library. Figure 4.1 provides an illustration of all the steps our solver has to perform when determining the existence of a model of an input formula.

4.2 Symbolic representation of automaton transition relations

The LSBF alphabet grows exponentially with the number of variables present in the input formula. Consequently, the time complexity of the classical automaton algorithms iterating over all alphabet symbols grows exponentially as well. Although the classical constructions are usable when deciding formulae with a low number of variables, the decision procedure based on these algorithms becomes quickly unfeasible when executed on richer formulae having several variables. The exponential growth of the subset construction is portrayed in Figure 4.2. Moreover, storing transitions explicitly represents a considerable portion of the memory used by the solver, as any transition over a symbol $s \in \Sigma$ will be stored with its own copy of the symbol. We have experimented with multiple ways to address these issues by representing the transitions symbolically.

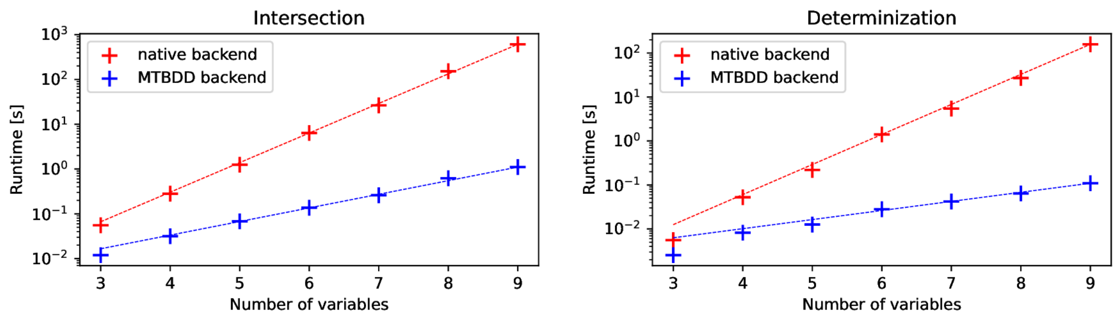
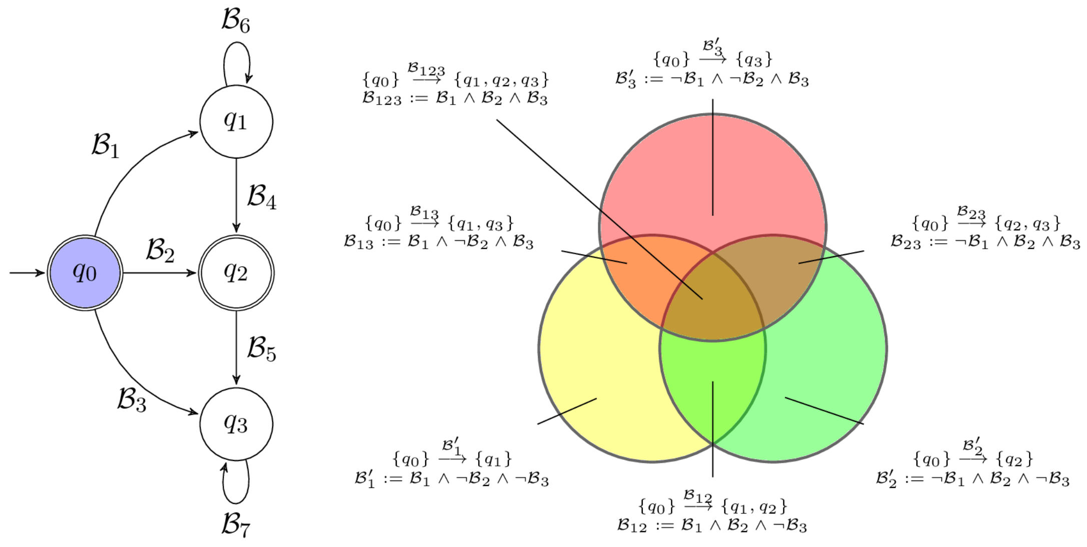


Figure 4.2: The exponential runtime growth of the classical constructions (native backend) wrt. the number of variables compared to the runtime of our optimized MTBBD-based procedures (MTBDD backend).

When searching for a performant symbolic representation, Binary Decision Diagrams (BDDs) presented themselves as a natural first choice due to their popularity and the availability of multiple mature Python libraries providing an optimized BDD implementation. Our BDD-based representation of the transition relation used one BDD to encode the transition symbols between every pair of two states. Although initially showing promising increase in the performance of the intersection procedure, determinization suffered significantly due to the complexity of evaluating minterms. The minterminization problem is depicted in Figure 4.3. The standard subset construction would begin by processing the macrostate $\{q_0\}$. This macrostate can have up to seven reachable successor macrostates ($|\mathcal{P}(Post(\{q_0\}))|$), and, for every of these possibly reachable macrostates, the solver has to compute a different BDD representing the transition symbols. If the calculated BDD represents $\bot$, the macrostate is discarded, as it is unreachable. Solving this problem for a macrostate with more than three reachable states in the input NFA quickly becomes more costly than iterating over the entire alphabet.

After failing to overcome performance problems connected to determinization of NFAs having transition symbols represented by BDDs, we shifted our attention to MTBDDs. Since MTBDDs can have leaves with arbitrary values, we were able to design a representation of the transition relation where every automaton state q has associated an MTBDD that represents all transitions from q. An illustration of the developed MTBDD representation is given in Figure 4.4. The idea of using MTBDDs to represent transition relations is not novel, and MTBDD-based representations been utilized by e.g. the VATA library [22] and the MONA solver [19]. In contrast to our earlier attempts representing symbolically only the transition symbols, the MTBDD-based approach represents symbolically entire sets of transitions.

MTBDDs are less popular compared to BDDs, lacking of any Python library implementing this formalism. In order for our solver to utilize this formalism, we had to create our own Python wrapper around Sylvan [11] — a mature BDD library implementing various BDD variants including MTBDDs. Sylvan was created as a research project aiming



$\mathcal{B}_6$
q_1
$\mathcal{B}_1$
$\mathcal{B}_4$
q_0
$\mathcal{B}_2$
q_2
$\mathcal{B}_5$
$\mathcal{B}_3$
q_3
$\mathcal{B}_7$
$\{q_0\} \xrightarrow{\mathcal{B}_{123}} \{q_1, q_2, q_3\}$
$\mathcal{B}_{123} := \mathcal{B}_1 \wedge \mathcal{B}_2 \wedge \mathcal{B}_3$
$\{q_0\} \xrightarrow{\mathcal{B}'_3} \{q_3\}$
$\mathcal{B}'_3 := \neg\mathcal{B}_1 \wedge \neg\mathcal{B}_2 \wedge \mathcal{B}_3$
$\{q_0\} \xrightarrow{\mathcal{B}_{13}} \{q_1, q_3\}$
$\mathcal{B}_{13} := \mathcal{B}_1 \wedge \neg\mathcal{B}_2 \wedge \mathcal{B}_3$
$\{q_0\} \xrightarrow{\mathcal{B}_{23}} \{q_2, q_3\}$
$\mathcal{B}_{23} := \neg\mathcal{B}_1 \wedge \mathcal{B}_2 \wedge \mathcal{B}_3$
$\{q_0\} \xrightarrow{\mathcal{B}'_1} \{q_1\}$
$\mathcal{B}'_1 := \mathcal{B}_1 \wedge \neg\mathcal{B}_2 \wedge \neg\mathcal{B}_3$
$\{q_0\} \xrightarrow{\mathcal{B}'_2} \{q_2\}$
$\mathcal{B}'_2 := \neg\mathcal{B}_1 \wedge \mathcal{B}_2 \wedge \neg\mathcal{B}_3$
$\{q_0\} \xrightarrow{\mathcal{B}_{12}} \{q_1, q_2\}$
$\mathcal{B}_{12} := \mathcal{B}_1 \wedge \mathcal{B}_2 \wedge \neg\mathcal{B}_3$

Figure 4.3: A Venn diagram illustrating a single iteration of the subset construction (cf. Algorithm 2) processing the macrostate $\{q_0\}$ of an NFA with transition symbols represented by Binary Decision Diagrams

at increasing the performance of BDD operations by exploiting parallelism. However, for the purposes of designing and experimenting with a MTBDD-based representation of automata transition relation, the parallelization provided by Sylvan is not needed, and it is turned off, avoiding the need to implemented synchronization mechanisms throughout the solver. Although Sylvan supports user-defined MTBDD leaves and operators, implementing a rich wrapper allowing to define custom leaves from Python posed a challenge caused, e.g., by the Sylvan API design requiring to pass function pointers to housekeeping functions when defining custom leaf types. Instead, we created a library implementing the custom leaves in C++ and provided a thin Python wrapper for interacting with the custom library.

Interfacing with a statically typed language also required restricting the way automata are represented. Algorithms used throughout the decision procedure often produce automata with states having special semantics. For example, Procedure 5 produces an automaton with states labeled by integers relating the states to the original atomic constraint, and a single final state labeled differently, since its semantics is not related to the original atomic constraint the same way the states labeled by integers are. As Python is dynamically typed, the core automata library did not enforce any restrictions on the state labels, and therefore, the processed automata kept the semantics of their states. Although such a relaxed automaton representation is useful for inspecting and reasoning about the inner workings of the decision procedure, it becomes an inconvenience when interfacing with a statically typed language. Therefore, automata states are restricted to be only integers and their semantics can be optionally tracked as a part of the automaton metadata if enabled by the user.

4.2.1 Efficient automata algorithms using MTBDD-based transition relation representation

In order to fully utilize the symbolic representation of transitions that MTBDDs provide, most of the classical automaton algorithms needed to be redesigned in a fashion avoiding ex-

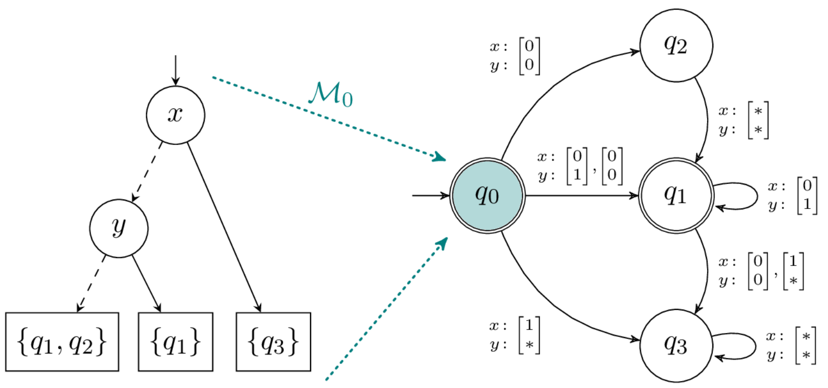

x
y
$\{q_1, q_2\}$
$\{q_1\}$
$\{q_3\}$
$\mathcal{M}_0$
q_0
q_1
q_2
q_3
$x: \; y: \begin{bmatrix}0\\0\end{bmatrix}$
$x: \; y: \begin{bmatrix}0\\1\end{bmatrix}, \begin{bmatrix}0\\0\end{bmatrix}$
$x: \; y: \begin{bmatrix}1\\*\end{bmatrix}$
$x: \; y: \begin{bmatrix}*\\*\end{bmatrix}$
$x: \; y: \begin{bmatrix}0\\1\end{bmatrix}$
$x: \; y: \begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}1\\*\end{bmatrix}$
$x: \; y: \begin{bmatrix}*\\*\end{bmatrix}$

Figure 4.4: An illustration of the MTBDD encoding all outgoing transitions of the state q_0 of the depicted NFA $\mathcal{A}$, where $*$ represents a don't care bit, dashed edges represent 0, and solid edges represent 1.

plicit iteration over the alphabet symbols, as such an iteration translates to uncompressing the MTBDD representation, and therefore, diminishing the potential performance gains of using this formalism.

The classical construction in Algorithm 2.2 of an automaton acepting union of two languages represented by corresponding automata does not require any modification. However, the automata library must make sure the set of states of both input automata are disjoint. This requirement is satisfied by renaming states of the input automata by assigning them unique integers, and therefore, the MTBDD library has to provide an unary state-renaming operator parametrized by a bijection mapping the old state names to the new names.

In contrast to the union construction, the classical intersection construction described in Algorithm 1 iterates over the automaton alphabet. In order to avoid such an iteration, a new MTBDD operator had to be designed. This operator achieves the same as the iteration, however, in a way that utilizes the MTBDD structure.

Let $\mathcal{A}_A = (Q_A, \Sigma, \delta_A, Q_{0A}, F_A)$ and $\mathcal{A}_B = (Q_B, \Sigma, \delta_B, Q_{0B}, F_B)$ be the input automata of the intersection procedure, let $\mathcal{A}_\cap = (Q_\cap, \Sigma, \delta_\cap, Q_{0\cap}, F_\cap)$ be the corresponding output automaton, and let $\diamond_\cap : \mathcal{P}(Q_A) \times \mathcal{P}(Q_B) \to \mathcal{P}(Q_A \times Q_B)$ be defined as $\diamond_\cap : (Q'_A, Q'_B) \mapsto Q'_A \times Q'_B$. The definition of the function $\diamond_\cap$ is based on the observation of the core principle of the intersection procedure. Let $(q, s) \in Q_\cap$ be a state of $\mathcal{A}_\cap$ where $q \in Q_A$ and $s \in Q_B$. Then the automaton $\mathcal{A}_\cap$ has transitions $Post_{\mathcal{A}_\cap}((q, s), \zeta) = Post_{\mathcal{A}_A}(q, \zeta) \times Post_{\mathcal{A}_B}(s, \zeta)$ for any $\zeta \in \Sigma$. In other words, the construction adds transitions from (q, s) along some symbol ζ based on the existence of transitions along the same transition symbol from the states q and s in the corresponding automata. The algorithm *BDDApply* (Algorithm 3) behaves in a similar fashion, applying the given function $\diamond$ on leaves reachable in the input MTBDDs by paths corresponding to the same Boolean vector. This similarity between the intersection procedure and the *BDDApply* algorithm results in the given definition of $\diamond_\cap$, that, when applied to the MTBDDs corresponding to the pair of states (q, s) currently processed in the intersection procedure, results in an MTBDD representing the transitions from (q, s) in automaton $\mathcal{A}_\cap$. An illustration $\diamond_\cap$ as used in the MTBDD-based intersection procedure can be seen in Figure 4.5.

However, the definition of $\diamond_\cap$ poses some implementation problems. MTBDDs used in the automata have leaves containing sets of states. The application of $\diamond_\cap$ results in an MTBDD with tuples of states stored in leaves, and therefore, it would require defin-

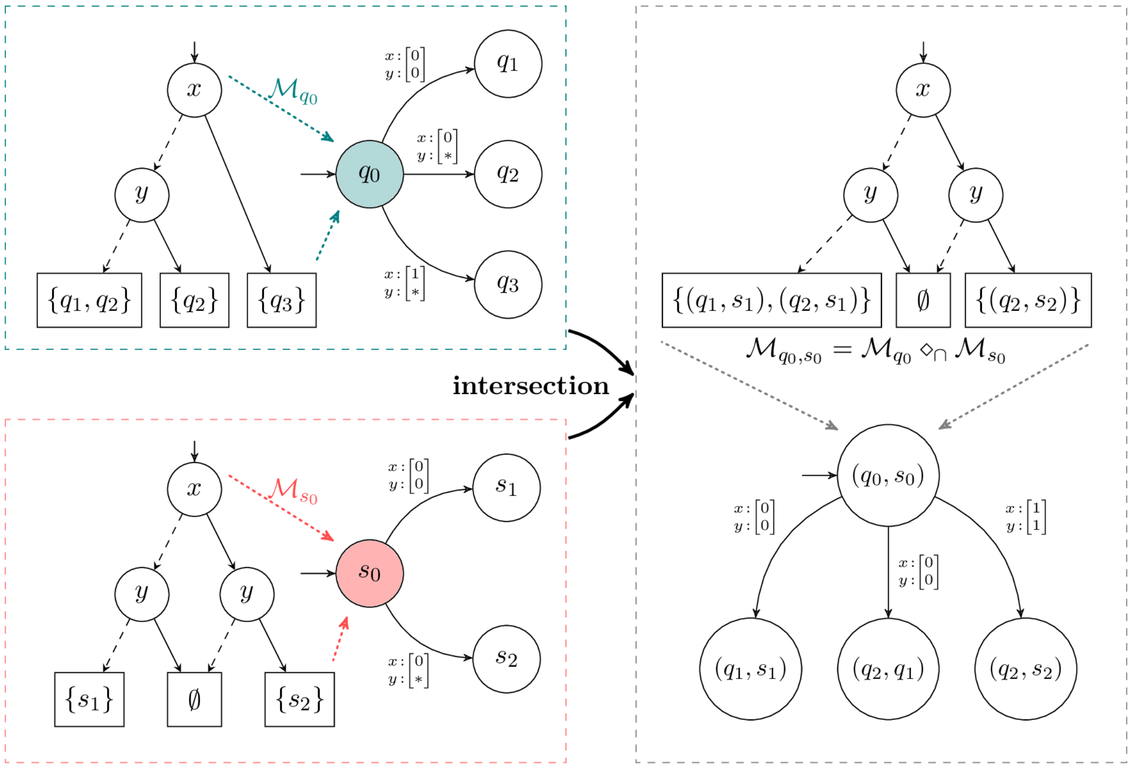


Figure 4.5: An illustration capturing a single step of the MTBDD-based intersection procedure processing the product state (q_0, s_0). The individual states constituting are displayed with relevant part of their automata, with the corresponding MTBDDs attached to the state. The MTBDD resulting from applying the intersection operator is also displayed, including the part of the automaton it represents.

ing a new leaf type in Sylvan, supplying Sylvan with custom housekeeping functions for the new leaf type, and exposing an interface allowing manipulation of MTBDDs with such leaves. Instead, our C++ library assigns the state tuples a unique integer when they are produced, avoiding the need to add a new leaf type definition while maintaining consistency of the type of leaves used by the automata at the same time. Any state of $\mathcal{A}_\cap$ might be reachable from multiple states, meaning that the same state tuple will be present in different Cartesian products, the assigned integers must be consistent throughout the MTBDD-based intersection procedure. Therefore, the implementation of $\diamond_\cap$ stores the information about assigned integers in a global context of the currently performed intersection, and propagates it to back to Python, keeping the information consistent between the Python side driving the intersection construction and the C++ side manipulating the MTBDDs.

Similarly to the intersection procedure, the subset construction also iterates over the automaton alphabet, and therefore, needs to be redesigned in order to utilize MTBDDs. Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ be the input automaton to be determinized, and let $\mathcal{A}^\mathcal{D} = (\mathcal{Q}, \Sigma, \Delta, \mathcal{Q}_0, \mathcal{F})$ be the corresponding equivalent DFA. The subset construction produces an automaton with states being sets of states of $\mathcal{A}$ and having transitions $M \xrightarrow{\zeta} M' \in \Delta$ where $M' = \cup_{q \in M}\{q' \mid q \xrightarrow{\zeta} q' \in \delta\}$. Similar to the intersection procedure, the transitions of $\mathcal{A}^\mathcal{D}$ are constructed based on the transitions from individual states in the macrostate M sharing the same transition symbol in the input automaton. This observation leads

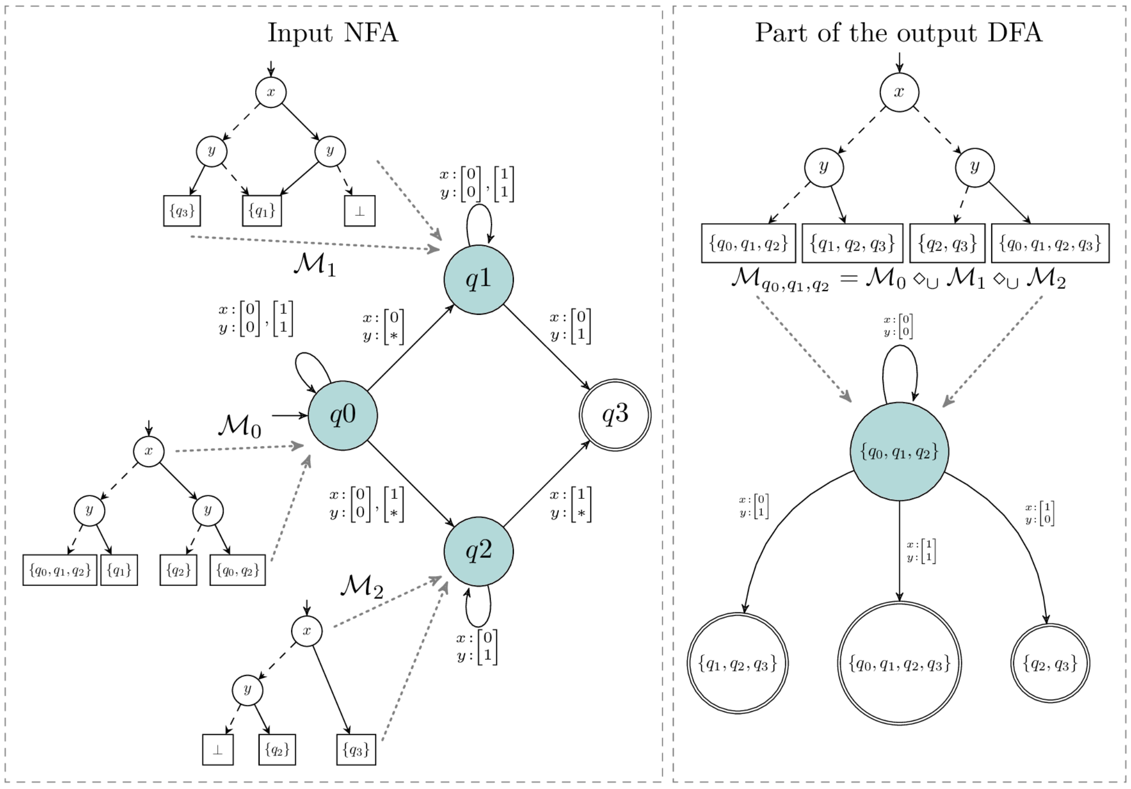


Figure 4.6: Illustration of a single step in the MTBDD-based subset construction processing the macrostate $\{q_0, q_1, q_2\}$.

to defining a new function $\diamond_\cup : \mathcal{P}(Q) \times \mathcal{P}(Q) \to \mathcal{P}(Q)$ as $\diamond_\cup : (Q_1, Q_2) \mapsto Q_1 \cup Q_2$. Given a macrostate $M = \{q_0, q_1\}$ and MTBDDs $\mathcal{M}_0$ and $\mathcal{M}_1$ representing transitions from the corresponding states in the input NFA, the application $\mathcal{M}_0 \diamond_\cup \mathcal{M}_1$ is analogous to the subset construction computing the transitions from M. As set union is associative, the MTBDD $\mathcal{M}_M$ representing transitions from a macrostate M can be computed by a consequent application of $\diamond_\cup$: $\mathcal{M}_M = (\mathcal{M}_0 \diamond_\cup \mathcal{M}_1) \diamond_\cup \dots \mathcal{M}_{|M|}$. Figure 4.6 provides an illustration of a single iteration of the MTBDD-based subset construction.

The resulting MTBDD $\mathcal{M}_M$ contains only sets of states in its leaves, and, therefore, it does not require defining a new leaf type. However, as the sets of states contained in MTBDD leaves are consider states of the output DFA, the MTBDD-based determinization must be finalized by replacing all sets in the MTBDD leaves by a singleton containing an unique integer representing the state in the constructed DFA in order to maintain the consistency of the MTBDD-based transition relation representation.

In contrast to the subset construction and the intersection construction, the *PadClosure* procedure (Algorithm 7) does not operate on the input automaton in a typical iterative way, in which a list of states to be processed is maintained, and only one state is processed at a time. Instead, the *PadClosure* algorithm computes a set S_ζ containing all states q such that $\mathcal{L}(q) \cap \zeta^+ \neq \emptyset$ for every $\zeta \in \Sigma$. In other words, the set S_ζ contains all states that accept a word consisting only of the padding symbol, and, therefore, the set contains all states possibly breaking the saturation property wrt. the symbol ζ. Computing S_ζ from automaton with transitions represented by MTBDDs while utilizing the MTBDD structure to avoid enumeration of transition symbols is a harder problem than designing the previous

MTBDD variants of the classical constructions due to the difference in their structures. Theoretically, a MTBDD-based *PadClosure* could use MTBDDs to efficiently partition the transition function of the input automaton according to the transition symbols. Entire automaton would then be represented by one MTBDD containing every equivalence class of such partitioning in one of its leaves. Iterating over the leaves of such an MTBDD and computing the set S from the equivalence classes is then a trivial task. However, this scheme would require adding another custom leaf type, further broadening the scope of the interface required between Python and our C++ library implementing functions for manipulating MTBDDs.

Instead, we created Algorithm 8, an iterative version of the *PadClosure* algorithm similar to the classical constructions. Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ be the input automaton for the *MTBDDPadClosure* procedure. The algorithm processes tuple of states (q, s) such that $s \in Post(q)$ and their corresponding MTBDDs. Starting with states with transitions to an accepting state and their predecessors, it attempts to propagate a transition to the final state if missing. The propagation is performed by the function $\diamond^*_{q_f} : \mathcal{P}(Q) \times \mathcal{P}(Q) \to \mathcal{P}(Q \cup \{q_f\})$, which, when supplied to *BDDApply*, adds a transition to a new final state q_f. The function $\diamond^*_{q_f}$ is defined as follows:

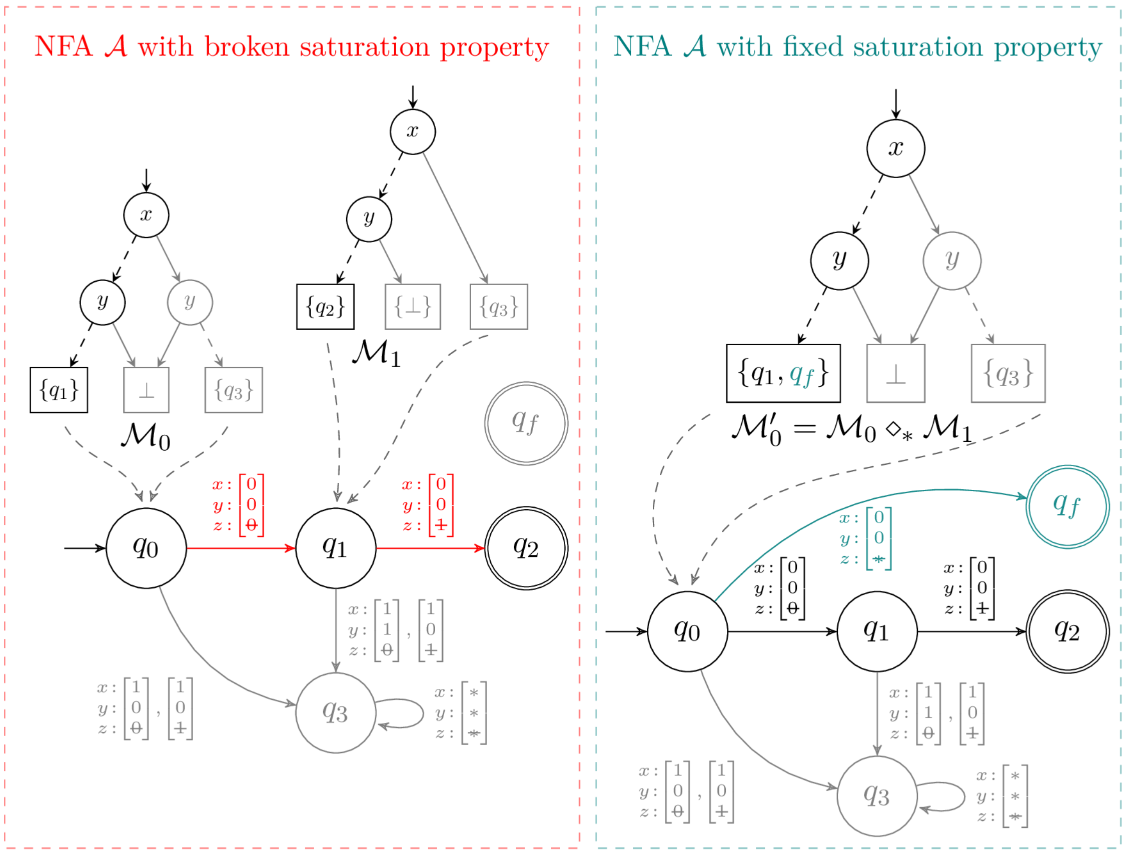
$$\diamond^*_{q_f}(Q_1, Q_2) = \begin{cases} Q_1 \cup \{q_f\} & Q_2 \cap F \neq \emptyset \wedge Q_1 \cap F \neq \emptyset \\ Q_1 & \mathit{otherwise} \end{cases}$$

If a transition to the new final state q_f was added, all predecessors of s are added to the work list, as the newly added transitions to a final state needs to be propagated further. In order to avoid reversing the MTBDDs, it is sufficient to compute *Pre* as an adjacency matrix at the beginning of the algorithm, by simply iterating through the leaves of the MTBDDs constituting δ. The algorithm is guaranteed to terminate, since by definition of $\diamond^*_{q_f}$ every state can be processed at most $|\Sigma|$ times, and the number of automaton states is finite. Figure 4.7 illustrates the usage of $\diamond^*_{q_f}$ during the *MTBDDPadClosure*.

Algorithm 8 Augment $\mathcal{A}$ with transition relation represented by MTBDDs so that it accepts all encodings of models.

```
Input: NFAs $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$
Output: NFA $\mathcal{A}' = (Q', \Sigma, \delta', Q_0, F')$ accepting all encodings of models
 1: function MTBDDPadClosure($\mathcal{A}$)
 2:     $\delta' \leftarrow \delta$
 3:     $W \leftarrow \emptyset$
 4:     $q_f \leftarrow$ a new state to be added to $\mathcal{A}$ such that $q_f \notin Q$
 5:     for $q \in \cup_{f \in F} Pre(f)$ do
 6:         for $s \in Pre(q)$ do
 7:             $W \leftarrow W \cup \{(q, s)\}$
 8:         end for
 9:     end for
10:     while $W \neq \emptyset$ do
11:         pick and remove $(q, s)$ from $W$
12:         $\mathcal{M}_q \leftarrow$ MTBDD corresponding to $q$ in $\delta'$
13:         $\mathcal{M}_s \leftarrow$ MTBDD corresponding to $s$ in $\delta'$
14:         $\mathcal{M}_{result} \leftarrow \mathcal{M}_q \diamond^*_{q_f} \mathcal{M}_s$
15:         if $\mathcal{M}_{result} \neq \mathcal{M}_q$ then
```

```
            for $s' \in Pre(s)$ do
                replace $\mathcal{M}_q$ in $\delta'$ by $\mathcal{M}_{result}$
                add $(s, s')$ to $W$ if $(s, s') \notin W$
            end for
        end if
    end while
    return $(Q, \Sigma, \delta, Q_0, F)$ if $\delta = \delta'$ else $(Q \cup \{q_f\}, \Sigma, \delta', Q_0, F \cup \{q_f\})$
end function
```



NFA $\mathcal{A}$ with broken saturation property
NFA $\mathcal{A}$ with fixed saturation property
$\mathcal{M}_0$
$\mathcal{M}_1$
$\mathcal{M}'_0 = \mathcal{M}_0 \diamond_* \mathcal{M}_1$
$\{q_1\}$
$\{q_2\}$
$\{q_3\}$
$\{q_1, q_f\}$
$\perp$
q_0
q_1
q_2
q_3
q_f

Figure 4.7: Demonstration of MTBDD-based padding closure

4.3 Additional features of our solver

Because the primary purpose of Amaya is experimantation with the automata-based decision procedure for PrA this purpose is also reflected in the additional features the solver provides.

To accurately compare the implemented decision procedure to the other approaches utilized by the state-of-the-art solvers, our solver implements a built-in benchmarking mechanism. Having a built-in benchmarking allows measuring the runtime of the decision procedure without including, e.g., the time consumed by the Python interpreter parsing the solver's codebase. Both Z3 and CVC4 provide a similar way to print statistics, including the used CPU time. The implemented benchmarking functionality allows specifying

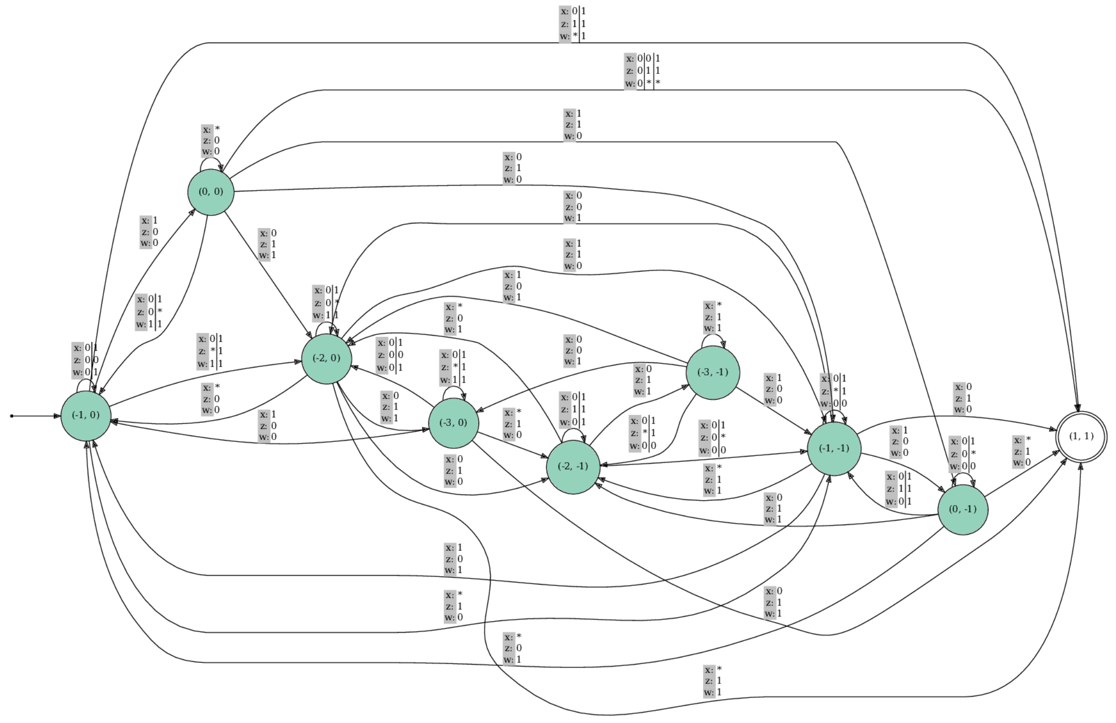


Figure 4.8: An example of an automaton constructed during the execution of the decision procedure as exported by Amaya in the DOT format. The automaton contains only one strongly connected component (SCC) with more than one state. The SCC is automatically detected by Amaya and its states are drawn in green.

the entire set of input files to benchmark the solver with, either by explicitly enumerating the desired input files or by supplying a folder containing the input formulae, letting the solver collect the files. Naturally, it is possible to specify the number of benchmark executions to minimize statistical noise. The solver then outputs the individual runtimes, as well as their average and the standard deviation in the JSON or the CSV format. The solver's benchmarking infrastructure also monitors whether the solver's answer matches the SAT value provided as a part of the input file, generating a report containing tests for which the computed results were incorrect, if any.

Amaya also provides a full introspection inside the decision procedure by exporting intermediate automata constructed during the decision procedure's run. The automata can be exported in the DOT format, providing an easy way to visualize and analyze the automata, or in the VTF format, making Amaya a source of automata to other research efforts in the field of automata theory. As the automata exported in the DOT format are usually inspected by a human, Amaya compresses the transition symbols using BDDs to provide a visual representation that is easier to read. An example of an exported automaton can be seen in Figure 4.8. Our solver can also optionally detect strongly connected components of the exported automaton and use this information to colorize the exported DOT representation, highlighting structures potentially interesting for further research. The export functionality is implemented in a way that is easy to extend, abstracting away the differences between the different automata representations used by the provided execution backends.

Parts of the codebase of our solver are covered by unit tests, providing a quick feedback on whether the algorithmic changes performed as a part of experimentation did not break the solver. These tests are written in a fashion asserting the automaton structure without making assumptions on the used state labeling, as the state labels may be different between various execution backends.

Chapter 5

Experimental evaluation

This chapter provides an overview of the results of experiments concluded with Amaya. The chapter starts by giving a comparison of our solver to the state-of-the-art solvers Z3 and CVC5 on real-world benchmarks with origins in program verification, as well as an evaluation of the performance benefits of the MTBDD-backend when deciding these benchmarks. The chapter then concludes by comparing our solver to the state of the art in deciding the Frobenius coin problem. As the presented results show, the automata-based decision procedure vastly outperforms the state of the art on the Frobenius coin problem benchmark. The experiments were concluded using the Z3 solver ver. `4.8.16` and the CVC5 solver ver. `1.0.0`, and all runtimes were measured with the precision of 1ms. All formulae included in our experiments contained information about their satisfiability in their metadata. The provided SAT values were used to check the computed SAT value.

The experiments were carried out on a system with the following configuration:

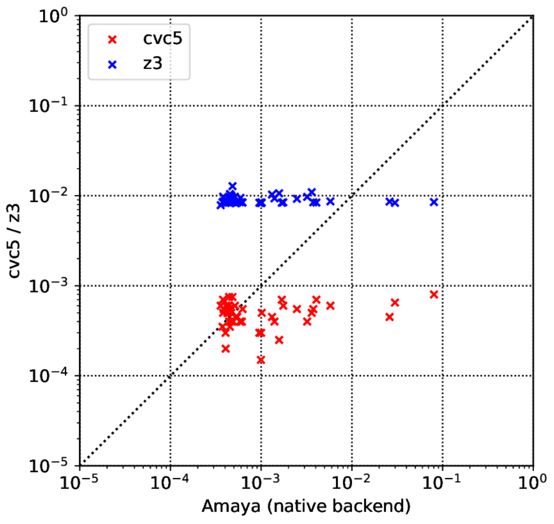
Machine	`Lenovo ThinkPad X1 Carbon Gen 5`
OS	`Debian GNU/Linux 11`
Kernel version	`#1 SMP Debian 5.10.106-1 (2022-03-17)`
CPU	`Intel(R) Core(TM) i7-7600U CPU @ 2.80GHz`
Cache	`128KiB L1, 512KiB L2, 4 MiB L3`
RAM	`16GB @ 1866MHz`
SSD	`512GiB M.2 SSD 32Gb/s`

5.1 Comparison to the state of the art on deciding quantified linear integer arithmetic benchmarks

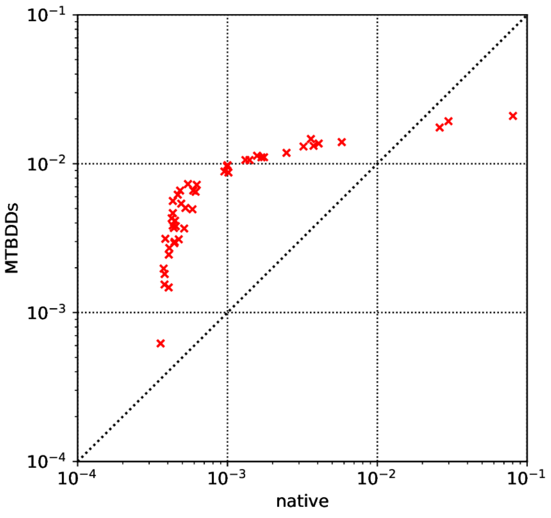
We have compared our solver to the state-of-the-art SMT solvers Z3 and CVC5 on deciding the TPTP and the UltimateAutomizer benchmarks from SMT-COMP available at the StarExec server[1].

The *TPTP (Thousands of Problems for Theorem Prover) Problem Library* [25] is a project aiming at providing a common library of problems to test and evaluate automated theorem proving systems. The problems contained in TPTP belong to numerous theories, including 46 linear integer arithmetics (LIA) formulae. Figure 5.1a depicts the runtimes of Amaya compared to the state-of-the-art solvers. As the results show, the state-of-the-art solvers decide the TPTP formulae with runtimes close to the measured time units. The low

[1] Available at: `https://www.starexec.org/starexec/secure/explore/spaces.jsp?id=446562`



(a) Comparison with the state of the art

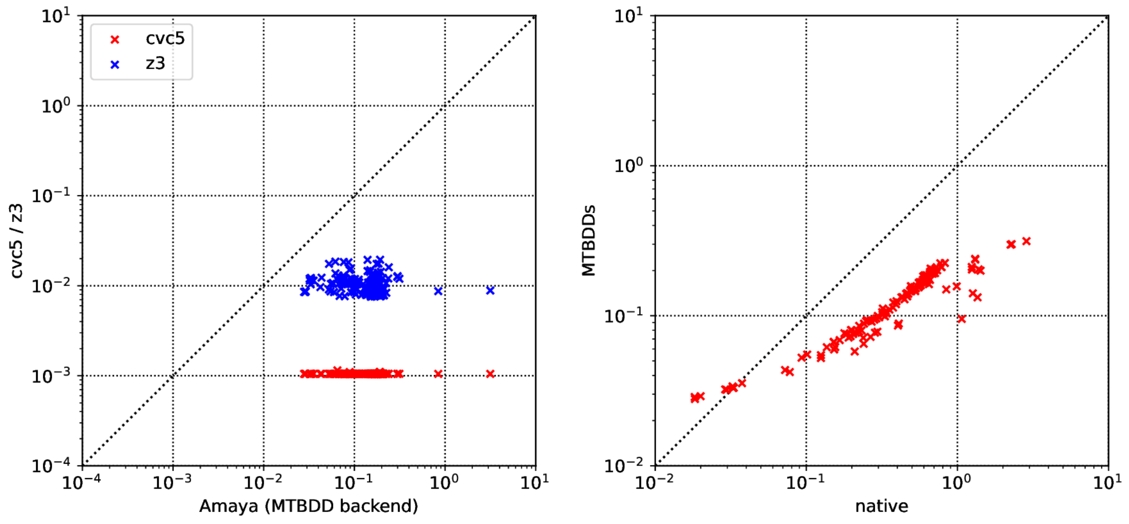


(b) Backend comparison

Figure 5.1: Comparison of our solver to the state of the art, as well as the comparison of the execution backends provided by our solver in deciding the TPTP LIA problems

runtimes are due to TPTP not containing complex formulae with many variables or atomic constraints with large coefficients. However, TPTP formulae contain all the Boolean connectives and quantifiers, including formulae with quantifier alternation. The low complexity combined with the spectrum of Boolean connectives makes TPTP a convenient benchmark to verify that the solver is providing correct answers. Therefore, the TPTP formulae serve as a set of integration tests for Amaya. Figure 5.1b compares the two execution backends our solver provides. The results show that the MTBDD-based backend is slower than the native backend, which stores transition symbols explicitly. This speed difference is due to the overhead of the `ctypes` library interface between C++ and Python. The observed overhead is caused by the need to serialize and deserialize data with every call to Amaya's C++ library, frequently allocating temporary memory.

We have also compared Amaya to the state of the art in solving the UltimateAutomizer benchmark, containing formulae generated by *UltimateAutomizer* [18] — an automatic verification tool for C programs. As the problems found in the UltimateAutomizer benchmark originate from real-world software verification, their complexity is much higher than the complexity of the problems found in TPTP. The results of the performance comparisons in deciding the UltimateAutomizer benchmark are presented in Figure 5.2. Naturally, our experimentation-oriented implementation written in an interpreted language is slower than the state-of-the-art solvers (cf. Figure 5.2a). However, the increased complexity of the input formulae is reflected in the depicted comparison of the two execution backends our solver implements. The costs of interfacing with our C++ library are outweighed by the benefits MTBDDs provide when deciding formulae with an increased number of variables (cf. Figure 5.2b).



(a) Comparison with the state of the art

(b) Backend comparison

Figure 5.2: Comparison of Amaya to the state of the art, as well as a comparison of the execution backends provided by Amaya in deciding the UltimateAutomizer LIA problems

5.2 The Frobenius coin problem

The Frobenius coin problem is a famous problem named after the German mathematician Ferdinand Georg Frobenius, who would occasionally put forth the problem of determining the highest possible sum not representable by coins of certain coprime denominations during his lectures. Although there exists an explicit formula [3] giving an exact solution when there are only coins with two denominations, explicit quantifier-free formulae for more than two coins are not known. The coin problem is not limited only to currency; its applications can be found, e.g., in Petri net analysis [9]. The monograph [2] provides an entire chapter dedicated to the applications of this problem. The coin problem can be written as the following formula, where f is the Frobenius number, $\vec{w}$ is a vector of pairwise coprime coin denominations:

$$Frob(f, \vec{w}) \triangleq \forall \vec{n} \in \mathbb{N}^{|w|} \colon (f \neq \vec{w} \cdot \vec{n}) \land (\forall f' \in \mathbb{N} \colon ((\forall \vec{n'} \in \mathbb{N}^{|w|}(f' \neq \vec{n'} \cdot \vec{w})) \rightarrow f' \leq f)) \tag{5.1}$$

To compare the performance of our solver to the state of the art on deciding the Frobenius coin problem, we have generated instances of $Frob(f, \vec{w})$ with two coins with denominations being consequent primes. Figure 5.3 shows that the automata-based decision procedure vastly outperforms the state-of-the-art solvers.

As shown in Figure 5.4, the state-of-the-art solvers struggle even when given the following simple formula $\forall z \in \mathbb{Z}(\exists x, y \in \mathbb{Z}(f = xw_1 + yw_2))$. The formula is structurally similar to a part of the formula encoding the solution of the Frobenius coin problem, and is satisfied iff a plane in a 3D space given by a normal vector (w_1, w_2) projected onto the z-axis covers all integer points of the axis.

The conducted experiments also provide insights into the bottlenecks of the automata-based decision procedure. The automata-based approach struggles when the size of the automata encoding equations and inequations have too many states due to large variable coefficients present in the corresponding atomic constraints, as the size of the automata

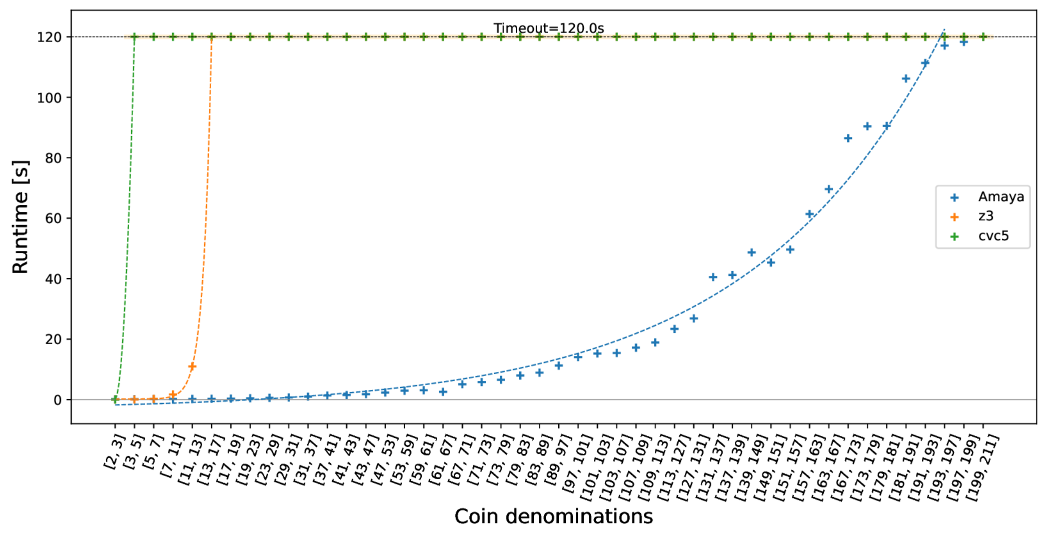


Figure 5.3: The runtime comparison of our solver to the state of the art on deciding the Frobenius coin problem

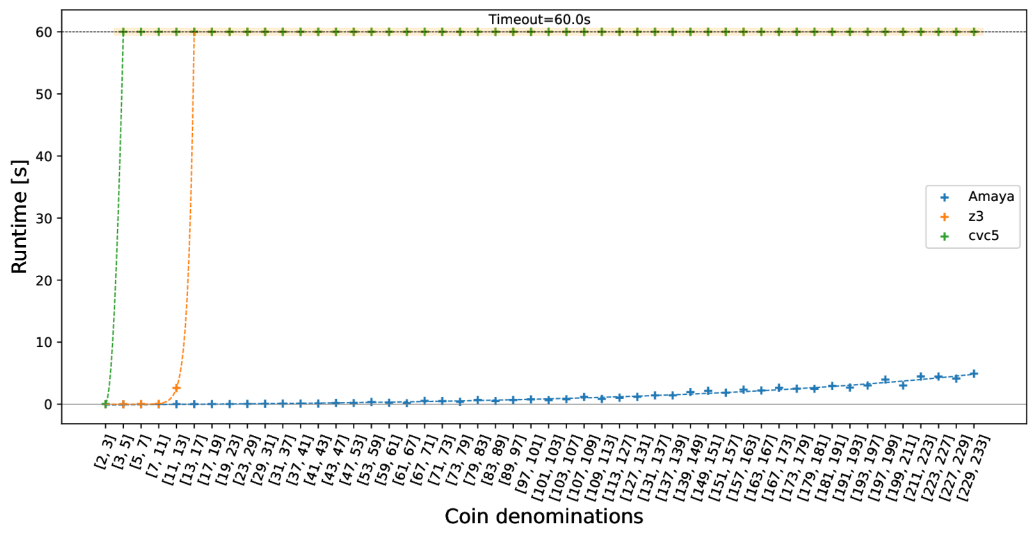


Figure 5.4: The runtime comparison of our solver to the state of the art on deciding whether a plane in a 3D space projected onto the z-axis covers all integer points of the axis

encoding such constrains is given by their variable coefficients [12]. Similarly, the size of automata encoding congruences is directly proportional to the divisor if the divisor is odd [12]. The automata-based decision procedure is unfeasibly costly when deciding even the simplest formulae containing such problematic constrains. The performance problems caused by big constants in atomic constraints could be probably alleviated by constructing intermediate automata lazily, avoiding eager construction of big automata.

Chapter 6

Conclusion

We have successfully created Amaya — an SMT solver for Presbuger arithmetic capable of deciding real-world problems based on the formal model of finite automata — an approach that no other current SMT solver employs. Having a functional implementation enables us to evaluate the performance characteristics of the automata-based procedure compared to the other approaches used by the state-of-the-art solvers. Being able to compare the different decision procedures allows us to identify problems in which automata present a faster alternative, pushing the limits of what we can decide automatically. To minimize the friction encountered when comparing different solvers, our solver supports a subset of SMT-LIB — a standardized input language — used to encode formulae of Presburger arithmetic.

The design of the presented implementation is oriented at providing an environment geared towards easy experimentation with the underlying decision procedure, and not aiming at besting the state-of-the-art solvers in terms of implementation efficiency. Therefore, our solver is written in the Python 3 programming language and is equipped with a range of features supporting experimentation. These features include an option to output all intermediate automata created during the decision procedure in various formats, including a human-friendly visualizable DOT language, or a built-in functionality allowing benchmarking the decision procedure on specified input problems in an easy manner.

After obtaining a minimal implementation, we identified scalability problems with some of the classical automata constructions e.g. determinization wrt. the number of variables in the input formulae. These problems are caused by the design of the classical constructions relying on iteration over the entire automaton alphabet, combined with the exponential growth of the alphabet when increasing the number of variables in the input formula. Therefore, we have crafted an automata representation storing the transition relation symbolically in the form of MTBDDs, in which every automaton state has assigned an MTBDD encoding the transitions from the corresponding state. All classical automata constructions had to be reformulated in a manner utilizing the compact representation of transitions that MTBDDs provide. As the ecosystem of the Python programming language did not provide an MTBDD implementation, we reached for Sylvan — a C library providing an optimized MTBDD implementation. In order for Amaya to use Sylvan, we had to implement the necessary MTBDD manipulation functions in C++, and to create a Python wrapper allowing us to interface with our C++ codebase. Therefore, Amaya provides two execution backends the user can choose from: an experimentation-oriented backend storing transitions explicitly, and an performance-focused backend storing transitions symbolically using MTBDDs.

By building a modern automata-based SMT solver for Presburger arithmetic, we have laid the necessary foundation required for future research in the practical applications of automata in the context of deciding Presbuger arithmetic. Our early experiments comparing the performance of Amaya to the state of the art showed that our experimentation-oriented implementation written in an interpreted language is slower than the state-of-the-art solvers. However, we have already managed to come across problems — e.g., deciding the Frobenius coin problem — in which automata present a much faster alternative, thus, highlighting the importance of the existence of a solver implementing an alternative approach.

6.1 Future work

Reaching a mature implementation capable of deciding non-trivial benchmarks does not signify that our work is concluded. On the contrary, a mature implementation presents a necessary basis for applying and evaluating the advancements in the field of automata theory to the implemented decision procedure. Namely, a lot of formulae from the UltimateAutomizer benchmark contain only one quantifier alternation, and, therefore, the generally expensive determinization after a track is projected away due to an existential quantifier could be avoided by using antichains to check for language non-universality [10]. To use antichains, our solver would have to be extended to support at least partially converting the formula into the prenex normal form. Another possibility is to use an augmented subset construction using the simulation preorder to prune some of the redundant states produced throughout the construction [15].

Our experimentation identified benchmarks containing problems with atomic constraints containing modulo terms with odd constants in the order of several hundred thousand. As there is no direct automaton construction for atomic constraints containing modulo terms, the modulo terms are rewritten using an existential quantifier as follows: $\vec{a} \cdot \vec{x} + (y \ mod \ M) \leq c \Leftrightarrow \exists m(\underbrace{\vec{a} \cdot \vec{x} + m \leq c \land 0 \leq m \leq M - 1}_{\mathcal{A}_\land} \land \underbrace{m \equiv_M y}_{\mathcal{A}_\equiv})$, with the problematic constant occurring only in the congruence part of the existential formula. The already-minimal automata encoding these congruence constraints have their number of states given approximately by the big constant, and therefore, their size causes the automata-based decision procedure to become prohibitively expensive. Furthermore, even the construction of such an automaton (cf. Algorithm 6) will take a considerable portion of the solver's runtime as the dynamic nature of Python causes the numerical calculations frequently performed during the construction of such an automaton to present a performance bottleneck. Whereas the latter can be addressed by implementing the construction in the C programming language and providing a Python wrapper, the automaton size remains a problem. Therefore, we would like to represent the automaton $\mathcal{A}_\equiv$ for the congruence symbolically, where we would construct explicitly only the automaton $\mathcal{A}_\land$ augmented with a counter symbolically representing a state of the automaton $\mathcal{A}_\equiv$.

Lastly, a possible direction to take is to apply the acquired knowledge about the architecture of an automata-based SMT solver and (at least partially) reimplement Amaya in a statically typed language such as C++. Such a new iteration of our solver would allow us to reduce the dimension of discussing the measured runtime differences between Amaya and the state-of-the-art solvers by removing the factor of the performance penalties inherited from a dynamic implementation language. A lower-level language would also allow us to optimize the solver to utilize more of the potential provided by modern computers.

Bibliography

[1] ABDULLA, P. A., CHEN, Y.-F., HOLÍK, L., MAYR, R. and VOJNAR, T. When Simulation Meets Antichains. In: ESPARZA, J. and MAJUMDAR, R., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, p. 158–174. ISBN 978-3-642-12002-2.

[2] ALFONSÍN, J. Applications of the Frobenius number. In: *The Diophantine Frobenius Problem*. New York: Oxford University Press, 2005, p. 159–184. Oxford Lecture Series in Mathematics and Its Applications. ISBN 978-0-191-52448-6.

[3] ALFONSÍN, J. The Frobenius number for small n. In: *The Diophantine Frobenius Problem*. New York: Oxford University Press, 2005, p. 31–44. Oxford Lecture Series in Mathematics and Its Applications. ISBN 978-0-191-52448-6.

[4] BARRETT, C., FONTAINE, P. and TINELLI, C. *The SMT-LIB Standard: Version 2.6* [online]. Department of Computer Science, The University of Iowa, 2017, 12.5.2021 [cit. 29.4.2022]. Available at: http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2021-05-12.pdf.

[5] BOUDET, A. and COMON, H. Diophantine equations, Presburger arithmetic and finite automata. In: KIRCHNER, H., ed. *Trees in Algebra and Programming — CAAP '96*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, p. 30–43. ISBN 978-3-540-49944-2.

[6] BRYANT, R. E. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*. IEEE. august 1986, C-35, no. 8, p. 677–691. DOI: 10.1109/TC.1986.1676819. ISSN 0018-9340.

[7] BÜCHI, J. R. Weak Second-Order Arithmetic and Finite Automata. In: MAC LANE, S. and SIEFKES, D., ed. *The Collected Works of J. Richard Büchi*. New York, NY: Springer New York, 1990, p. 398–424. DOI: 10.1007/978-1-4613-8928-6_22. ISBN 978-1-4613-8928-6.

[8] CHISTIKOV, D., HAASE, C. and MANSUTTI, A. Quantifier elimination for counting extensions of Presburger arithmetic. In: BOUYER, P. and SCHRÖDER, L., ed. *Foundations of Software Science and Computation Structures*. Cham: Springer International Publishing, 2022, p. 225–243. ISBN 978-3-030-99253-8.

[9] CHRZASTOWSKI WACHTEL, P. and RACZUNAS, M. Liveness of weighted circuits and the diophantine problem of Frobenius. In: ÉSIK, Z., ed. *Fundamentals of Computation Theory*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, p. 171–180. ISBN 978-3-540-47923-9.

[10] De Wulf, M., Doyen, L., Henzinger, T. A. and Raskin, J. F. Antichains: A New Algorithm for Checking Universality of Finite Automata. In: Ball, T. and Jones, R. B., ed. *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, p. 17–30. ISBN 978-3-540-37411-4.

[11] Dijk, T. van and Pol, J. van de. Sylvan: Multi-Core Decision Diagrams. In: Baier, C. and Tinelli, C., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, p. 677–691. ISBN 978-3-662-46681-0.

[12] Durand Gasselin, A. and Habermehl, P. On the Use of Non-deterministic Automata for Presburger Arithmetic. In: Gastin, P. and Laroussinie, F., ed. *CONCUR 2010 - Concurrency Theory*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, p. 373–387. ISBN 978-3-642-15375-4.

[13] Ernst, L. Succinct representation of regular languages by boolean automata. *Theoretical Computer Science*. 1981, vol. 13, no. 3, p. 323–330. DOI: 10.1016/S0304-3975(81)80005-9. ISSN 0304-3975.

[14] Esparza, J. *Automata theory, an algorithmic approach*. Technical University of Munich, august 2017 [cit. 4.5.2022]. Available at: `https://www7.in.tum.de/~esparza/autoskript.pdf`.

[15] Glabbeek, R. van and Ploeger, B. Five Determinisation Algorithms. In: Ibarra, B., ed. *Implementation and Applications of Automata*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, p. 161–170. ISBN 978-3-540-70844-5.

[16] Gödel, K. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*. Dec 1931, vol. 38, no. 1, p. 173–198. DOI: 10.1007/BF01700692. ISSN 1436-5081.

[17] Hague, M., Lin, A. W., Rümmer, P. and Wu, Z. Monadic Decomposition in Integer Linear Arithmetic. In: Peltier, N. and Sofronie Stokkermans, V., ed. *Automated Reasoning*. Cham: Springer International Publishing, 2020, p. 122–140. ISBN 978-3-030-51074-9.

[18] Heizmann, M., Christ, J., Dietsch, D., Ermis, E., Hoenicke, J. et al. Ultimate Automizer with SMTInterpol. In: Piterman, N. and Smolka, S. A., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, p. 641–643. ISBN 978-3-642-36742-7.

[19] Henriksen, J. G. et al. Mona: Monadic second-order logic in practice. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, p. 89–110. ISBN 978-3-540-48509-4.

[20] Klaedtke, F. Bounds on the Automata Size for Presburger Arithmetic. *ACM Trans. Comput. Logic*. New York, NY, USA: Association for Computing Machinery. apr 2008, vol. 9, no. 2. DOI: 10.1145/1342991.1342995. ISSN 1529-3785.

[21] Lee, C. Y. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal*. july 1959, vol. 38, no. 4, p. 985–999. DOI: 10.1002/j.1538-7305.1959.tb01585.x. ISSN 0005-8580.

[22] Lengál, O., Šimáček, J. and Vojnar, T. VATA: A Library for Efficient Manipulation of Non-deterministic Tree Automata. In: Flanagan, C. and König, B., ed. *Tools and Algorithms for the Construction and Analysis of Systems.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, p. 79–94. ISBN 978-3-642-28756-5.

[23] Patras, F. Axioms and Formalisms. In: *The Essence of Numbers.* Cham: Springer International Publishing, October 2020, p. 121–134. DOI: 10.1007/978-3-030-56700-2_12. ISBN 978-3-030-56700-2.

[24] Stansifer, R. *Presburger's Article on Integer Airthmetic: Remarks and Translation.* TR84-639. Cornell University, Computer Science Department, september 1984 [cit. 29.4.2022]. Available at: `https://hdl.handle.net/1813/6478`.

[25] Sutcliffe, G. The TPTP Problem Library and Associated Infrastructure. *Journal of Automated Reasoning.* february 2017, vol. 59, no. 4, p. 483–502. DOI: 10.1007/s10817-017-9407-7. ISSN 1573-0670.

[26] Turoňová, L., Holík, L., Lengál, O., Saarikivi, O., Veanes, M. et al. Regex Matching with Counting-Set Automata. *Proc. ACM Program. Lang.* New York, NY, USA: Association for Computing Machinery. nov 2020, vol. 4, OOPSLA. DOI: 10.1145/3428286.

Appendix A

Contents of the included storage media

```
/
├── amaya-python/ ...................... Amaya's implementation
│   ├── amaya/ ........................... Python module implementing Amaya's func-
│   │                                      tionality
│   ├── tests/ ........................... Unit tests
│   │   └── amaya-mtbdd.so ............... Precompiled library implementing MTBDD
│   │                                      operations
│   ├── README.md ........................ Installation instructions and usage examples
│   ├── requirements.txt ................. Amaya's Python dependencies
│   ├── run-amaya.py ..................... Main script for executing Amaya
│   └── LICENSE.txt ...................... Amaya's license
├── amaya-cpp/ ........................... C++ library implementing MTBDD opera-
│   │                                      tions
│   └── README.md ........................ Installation and compilation instructions
├── thesis-source/ ....................... LaTex sources of this thesis
└── xhecko02-thesis.pdf/ ................. Digital version of this thesis
```