Light On String Solving

Approaches to Efficiently and Correctly Solving String Constraints

Mitja Kulczynski

Dissertation
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.-Ing.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel
eingereicht im Jahr 2021

Kiel Computer Science Series (KCSS) 2022/1 dated 2022-01-17

ISSN 2193-6781 (print version)
ISSN 2194-6639 (electronic version)

Electronic version, updates, errata available via https://www.informatik.uni-kiel.de/kcss

The author can be contacted via mitja@kulczyn.ski

Published by the Department of Computer Science, Kiel University

Dependable Systems Group

Please cite as:

- Mitja Kulczynski. *Light On String Solving* Number 2022/1 in Kiel Computer Science Series. Department of Computer Science, 2022. Dissertation, Faculty of Engineering, Kiel University.

```
@book{Kulczynski22,
  author    = {Mitja Kulczynski},
  title     = {Light on String Solving},
  publisher = {Department of Computer Science, Kiel University},
  year      = {2022},
  number    = {2022/1},
  doi       = {10.21941/kcss/2022/1},
  series    = {Kiel Computer Science Series},
  note      = {Dissertation, Faculty of Engineering,
               Kiel University.}
}
```

About this Series

The Kiel Computer Science Series (KCSS) covers dissertations, habilitation theses, lecture notes, textbooks, surveys, collections, handbooks, etc. written at the Department of Computer Science at Kiel University. It was initiated in 2011 to support authors in the dissemination of their work in electronic and printed form, without restricting their rights to their work. The series provides a unified appearance and aims at high-quality typography. The KCSS is an open access series; all series titles are electronically available free of charge at the department's website. In addition, authors are encouraged to make printed copies available at a reasonable price, typically with a print-on-demand service.

Please visit `http://www.informatik.uni-kiel.de/kcss` for more information, for instructions how to publish in the KCSS, and for access to all existing publications.

1. Gutachter: Prof. Dr. Dirk Nowotka
Christian-Albrechts-Universität Kiel
Kiel, Deutschland

2. Gutachter: Prof. Dr. Stefan Göller
Universität Kassel
Kassel, Deutschland

Datum der mündlichen Prüfung: 17. Dezember 2021

Acknowledgements

> *"There's a science to walking through windows."*
>
> THE NATIONAL

This work would not have been possible without the support and encouragement of many people around me.

First of all, I am tremendously thankful to my supervisor Prof. Dr. Dirk Nowotka for his patience in finding my own way in academia. Without his suggestions, feedback and confidence in this work none of it would have been feasible.

I am deeply grateful to my collaborator Danny Bøgsted Poulsen for teaching me essentially everything to be a successful computer scientist. His invaluable advice was one of the reasons that kept me motivated during tough times.

I want to express my special thanks to Florin Manea who tirelessly supported me through my whole time as a Ph.D student. He lend me an ear whenever I needed it.

My deepest gratitude goes to Pamela Fleischmann for encouraging me to belief in my work, providing valuable feedback in any circumstances and making my teaching experience as amazing as it could get.

I also want to express thanks to Max Jonas Friese for unwavering support and feedback, as well as our nightly gaming sessions to ease stress.

Starting something new is never easy. I am indebted to Thorsten Ehlers who kept me going, showed me all required initial steps in academia, and was always there for a good chat and a beer.

Science is more than just writing papers. The time I spent with the Dependable Systems Group having lunch, playing board games or simply chatting in a break was invaluable. My thanks go to current and former group members: Philipp Sieweck, Lukas Haschke, Joel Day, Karoliina Lehtinen, and Yannik Potdevin.

I am very thankful to Murphy Berzish, Federico Mora, and Vijay Ganesh for our amazing work on Z3. Working with you was always extremely informative, productive and fun.

My gratitude is by Rudolf Berghammer. Thanks for encouraging me to starting my Ph.D and providing valuable feedback whenever I needed it.

Moreover, I want to appreciate the insightful comments of my secondary assessor Stefan Göller who let me see this work in a different light. Furthermore, I want to thank my examining committee, Michael Hanus and Thomas Wilke for their efforts.

Additionally, I want to thank all my co-authors not mentioned so far, that are Yannik Eikmeier and Axel Legay.

Getting through the tough times of this period would not have been possible without the support of my family. I am deeply thankful to Sabine, Uwe, Ute, Jörg, Dirk, Jui, Pietz, Eve, Jens, and Jenny. Friends are there to help you as it was the case for me. I heavily appreciate the support of Paul, Eva, Max, Anna, Casi, Nina, Marek, Daniel, Dean, and Luca.

Finally, I am deeply thankful to Vanessa who always supported me up to her limits. Not only saying the right things cheering me up again but also knowing how to proceed in certain situations – as watching Chelsea instead of working all night on a paper. Thanks for going with me through this journey making it a wonderful one.

Zusammenfassung

Die steigende Nutzung der formalen Analyse von Programmen, welche in modernen Programmiersprachen, die vielfach auf den Datentyp der Strings zurückgreifen, führt zur erweiterten Nachfrage an effizienteren und zuverlässigeren Methoden zur Lösung der entstehenden Formeln. Dies stellt insbesondere für das Lösen von industriellen Instanzen ein Problem dar. Die Entwicklung von Algorithmen für das (im Allgemeinen unentscheidbare) Entscheidungsproblem von Formeln in Logik erster Stufe über den Theorien der Wortgleichungen, linearen Ungleichungen und regulären Nebenbedingungen erfordert ein tiefgreifendes Verständnis jener Strukturen. In dieser Dissertation präsentieren wir einige Ansätze, um die vormals genannten Probleme zu lösen: Wir stellen einen Algorithmus vor, welcher das Erfüllbarkeitsproblem der beschränkten Wortgleichungen auf ein Erreichbarkeitsproblem über einem nichtdeterministischen endlichen Automaten reduziert. Anschließend beschreiben wir, wie die Erreichbarkeitsfrage im resultierenden Automaten in die Aussagenlogik überführt und mithilfe des SAT Lösers Glucose gelöst wird.

Weiterführend präsentieren wir ein Transformationssystem zum Lösen von Wortgleichungen, welches auf Basis eines aus der Theorie der Kombinatorik von Wörtern, weitgehend als Lemma von Levi bekannt, erstellt wird. Wir erweitern die induzierten Regeln um weitere, aus der Theorie bekannte klassische Lemmata, um Transitionsschritte zu vereinfachen. Des Weiteren bietet unser Transitionssystem nicht nur die Möglichkeit Wortgleichungen zu lösen, sondern auch das Lösen von linearen Ungleichungen, welche über den Variablen der Wortgleichungen formuliert sind. Die Effizienz dieses Ansatzes wurde weiter verbessert, indem wir externe Wortgleichungslöser bei Bedarf mittels Heuristiken hinzuziehen.

Im dritten Teil untersuchen wir die Struktur einer Menge von Instanzen, welche in verwandter Literatur vorgestellt wurde und reguläre Membership Constraints enthalten, und identifizierten verschiedenste Teiltheorien des untersuchten Fragments der String Constraints. Zu vielen der entdeckten Theorien konnten wir deren Entscheidbarkeit beziehungsweise Unentscheidbarkeit nachweisen. Für einige der resultierenden Theorien konnten wir sogar die PSPACE-Vollständigkeit beweisen. Hervorzuheben ist an dieser Stelle, dass die überwiegend aus kommerzieller Nutzung stammenden untersuchten Instanzen in eine der PSPACE-vollständigen Theorien einzubetten sind. Wir haben dieses Wissen genutzt und unsere Beweisstrategie in einem der meistgenutzten SMT Löser Z3STR3 implementiert.

Während der Auswertung unserer Methoden ist uns aufgefallen, dass es keine Möglichkeiten gibt, unsere Algorithmen mit bereits vorhandenen Lösern zu messen. Um dies zu ändern, haben wir die Literatur nach Instanzen, die in verschiedenen Szenarios zur Evaluation von Wortgleichungslösern verwendet wurden, durchsucht und diese gesammelt. Des Weiteren haben wir ein Werkzeug entwickelt, welches die faire Auswertung der individuellen Leistung der verschiedenen Löser über den gesammelten Instanzen ermöglicht. In dieser Arbeit nutzen wir unser Werkzeug, um die Effizienz, Zuverlässigkeit und Leistung der hier vorgestellten Methoden zu evaluieren.

Abstract

Widespread use of string solvers in formal analysis of string-heavy programs has led to a growing demand for more efficient and reliable techniques which can be applied in this context, especially for real-world cases. Designing an algorithm for the (generally undecidable) satisfiability problem for systems of string constraints requires a thorough understanding of the structure of constraints present in the targeted cases. In this thesis we target the aforementioned case in different perspectives: We present an algorithm which works by reformulating the satisfiability of bounded word equations (also called string equations) as a reachability problem for nondeterministic finite automata, and then carefully encoding this as a propositional satisfiability problem, which we then solve using the well-known GLUCOSE SAT-solver.

Secondly, we present a transformation-system-based technique to solving string constraints, by reformulating a classical combinatorics on words result, the lemma of Levi. We further enrich the induced rules by simplification steps based on results from the combinatorial theory of word equations, as well as by the addition of linear length constraints. This transformation-system approach cannot solve all equations efficiently by itself. To improve the efficiency of our transformation-system approach we integrate existing successful string solvers, which are called based on several heuristics.

Thirdly, we investigate benchmarks presented in the literature containing regular expression membership predicates, extract different first order logic theories, and prove their decidability, resp. undecidability. Notably, the most common theories in real-world benchmarks are PSPACE-complete. We use the strategy of the proof to design an efficient algorithm which was integrated into Z3STR3, one of the state of the art string solvers.

While evaluating our experimental implementations of the aforementioned approaches, we realised that a common framework to compare string solvers seems to be missing. To cope with this, we gathered a set of relevant benchmarks and introduce our new benchmarking framework to address this purpose. We used this framework to showcase the power of our algorithms via an extensive empirical evaluation over a diverse set of benchmarks.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> *"Raise your glass to the night time and the ways, to choose a mood and have it replaced."*
>
> BALTHAZAR

A word equation is a pair of strings (called the sides of the equation) consisting of symbols of two types: *constant* letters and *variables*. The word equation satisfiability problem is to determine whether we can unify the two strings, i.e., transform them into two equal strings containing constant letters only, by substituting the variables consistently by strings of constants. For example, consider the equation defined by the two strings $\mathsf{x}_1 ab \mathsf{x}_2$ and $a\mathsf{x}_1\mathsf{x}_2 b$, denoted $\mathsf{x}_1 ab \mathsf{x}_2 \doteq a\mathsf{x}_1\mathsf{x}_2 b$, with variables $\mathsf{x}_1, \mathsf{x}_2$ and constants a and b. It is satisfiable because x_1 can be substituted by a and x_2 by b, which produces the equality $aabb = aabb$. In fact, substituting x_1 by an arbitrary number of a and x_2 by an arbitrary number of b's unifies the two sides of the equation. As another example, the equation $a\mathsf{x}_1\mathsf{x}_1 \doteq \mathsf{x}_1 b \mathsf{x}_2$ is not satisfiable. A set (or system) of equations is satisfiable if there exists a substitution of the variables which solves all the equations simultaneously. In general we are concerned with the Σ_1, respectively quantifier free fragment of first order logic formulae having word equations as the only atom.

The satisfiability problem for word equations is an important problem in both mathematics and computer science. For instance, in an attempt to solve Hilbert's tenth problem [61] in the negative, Markov showed a reduction from word equations to Diophantine equations (see [70, 85, 87]), in the hope that word equations would prove to be undecidable. However, Makanin [87] proved in 1977 that the satisfiability of word equations *is*, in fact, decidable. Considerable effort has also been spent identifying the complexity of deciding the satisfiability of a word equation. After a series of intermediate results [85], Plandowski [94] showed that this problem is in PSPACE. In a series of recent papers [65, 66], Jeż applied a new elegant technique called recompression to word equations to show that word equations can be solved in non-deterministic linear space. However, there is a mismatch between the aforementioned upper bounds and the only known lower bound: solving word equations is NP-hard.

In more recent years, word equations have gained attention from the formal verification and security community, because word equations naturally occur during symbolic execution of high-level languages. A variety of analysis, testing, and verification methods have been proposed to address this problem [12, 26, 38, 49, 71, 96, 99, 103, 117], many of those depend on a tool which is able to solve word equations. These tools are commonly called string constraint solvers or, for short string solvers.

In practice, more functionality than just solving word equations is required, so the theory of word equations is often extended with multiple other theories. String solvers usually support a rich quantifier free first order logic theory over word equations, linear integer arithmetic over word length, and regular expression membership constraints. A quantifier free first order logic formula over these theories is called a string constraint. Due to the complexity of solving this combined theories and the undecidability of some of these extensions, no solver provides a complete algorithm. Prominently, the decidability of the theory of word equations enhanced with a length function remains a major open problem in theory, despite many attempts to solve it over the last 50 years [89]. Solely looking at the quantifier free theory of word equations and regular expression membership predicates is known to be decidable [85]. Unfortunately, it is not known whether the satisfiability problem for string constraints involving all aforementioned theories is decidable or not. However, already in the presence of other simple and natural constraints, like string-number conversion, this problem becomes undecidable (cf. [41]).

According to the literature *Satisfiability Modulo Theories* (SMT) solvers seem to be the most successful string solvers. The general idea of these solvers is integrating a sub solver for each particular theory, e.g. a solver which explicitly handles word equations. To ease handling of real-world inputs many string solvers support additional functions such as integer to string conversion and predicates such as string containment which are used within source code. These operations are commonly called higher order operations.

Despite their difficulty, much research has been done on techniques for solving string constraints obtained from many real-world analysis, testing, verification, and synthesis applications [49, 83, 99, 117]. Examples of such solvers are HAMPI [71], STRANGER [120], Z3SEQ [123], CVC4 [80], NORN [4], TRAU [2], S3 [114], and Z3STR3 [23], each with varying strengths and weaknesses. Precisely, because solving string formulae is hard in general, solver designers have come up with a diverse set of practical algorithms that incorporate a variety of tradeoffs. Some of these methods work well for pure word equations, but not so well for integer constraints over string length. Other methods work well for a mix of word equations and integer constraints, but perform poorly on more complicated constraints involving higher order operations. Given these weaknesses, within this work we enrich the area with new solving strategies, new theoretical insights, and to the best of our knowledge, the first framework for evaluating and testing string solvers.

This document is organised as follows: In Chapter 2 we introduce the formal background. After establishing the general notations used in this work, we present the underlying first order logic theories forming string constraints, as well as the commonly used higher order operations.

Chapter 3 continues embedding string constraints into their practical applications. Firstly, we motivate why string constraints are an important aspect when verifying real-world applications. We also show, how string constraints are derived based on real-world source code. We continue introducing the most used string solvers and their underlying procedures. This chapter closes with a presentation of two general solving strategies which heavily influence the procedures presented in this work.

In Chapter 4 we present a collection of techniques we created to solving string constraints. The first approach is built on top of a SAT solver encoding string constraints into a propositional logic formula. We continue elaborating a technique which solves word equations by building a transition system based of rules induced by the famous lemma of Levi [78].

We enrich the transition system by several heuristics to invoke an external solver to speed up the search. Afterwards, we analyse several string solving benchmarks with respect to regular expression membership constraints and identify sub theories. We prove the decidability, respectively undecidability of some of the revealed fragments. Furthermore, we use the proof for the decidability of one of the largest occurring fragments to design an algorithm which was directly implemented into the Z3str3 SMT solver.

Chapter 5 establishes a framework for comparing, evaluating, and debugging string solvers. We explain how we built this tool and explain the underlying mechanisms. Furthermore, we present a collection of benchmarks previously published in the literature which was incorporated into our framework. We close this chapter by evaluating our framework based on the needs of string solvers.

Specifically targeting the techniques estabilished in Chapter 4, we empirically evaluate the performance of our approaches using the benchmark framework introduced in Chapter 5 in Chapter 6. We showcase that even after several years of publishing our techniques, we built reliable techniques performing very well with respect to other string solvers.

Each chapter has a conclusion itself, but in Chapter 7 we conclude the overall picture of this work and name some future directions with respect to the presented approaches.

Chapter 2

Preliminaries

> *"You think you know somethin' you don't."*
>
> Sharon Van Etten

Within this chapter we fix the basic notion used. We start with an introduction of mathematical notations. Afterwards, we fix the terminology of propositional and first order logics. Moreover, we introduce particular logical theories considered in this work.

Let $\mathbb{N}$ be the set of natural numbers (including 0), $[n]$ be the set $\{ 0, 1, \dots, n \}$, and $[n]_0 = [n] \setminus \{ 0 \}$. Let $n \in \mathbb{N} \setminus \{ 0 \}$ and M a set. Let $M^n = \prod_{i \in [n]_0} M$ denote the set of all n-tuples. Furthermore, we have $M^0 = \{ () \}$. For an infinite set M and a finite subset $N \subset M$ let $N \ddot{\subset} M$ denote N's finiteness. For arbitrary sets M,N and a relation $r \subseteq M \times N$ we call r a partial function (denoted by $r : M \rightharpoonup N$) if for all $x \in M$ 1. there exists a unique $y \in N$ such that $r(x) = y$, 2. or $r(x)$ is undefined. Let $\mathsf{dom}(r) = \{ x \in M \mid \exists\, y \in N : r(x) = y \}$ denote the domain of r.

An *alphabet* Δ is a set of symbols, where $a \in \Delta$ are called *letters*. By Δ^* we denote the set of all finite words over Δ and let $\varepsilon \in \Delta^*$ denote the *empty word*. For $n \in \mathbb{N}$ let $w = a_1 \dots a_n \in \Delta^*$ such that $a_i \in \Delta$ for $i \in [n]_0$ be a word, i.e a finite sequence . By $w[i] = a_i$ we refer to the letter at the i^{th} position of w. Furthermore let $w[i : j] = a_i \dots a_j$ for $i, j \in [n]_0$ and $i \leqslant j$ refer to the *factor* starting at position i and ending at position j in w. Note, whenever $i = j$ we have $w[i : j] = w[i] = a_i$. Moreover, to ease readability we let $w[: j]$ denote $w[1 : j]$ and $w[i :]$ denote $w[i : n]$. Let $| \cdot | : \Delta^* \to \mathbb{N}$

defined by

$$|w| = \begin{cases} 0 & \text{if } w = \varepsilon, \\ 1 + |w[2:]| & \text{else.} \end{cases}$$

denote the *length* of a word w. Furthermore, let $|\cdot|_a : \Delta^* \to \mathbb{N}$ defined by

$$|w|_a = \sum_{i \in [|w|]_0} \begin{cases} 1 & \text{if } w[i] = a, \\ 0 & \text{else.} \end{cases}$$

denote the the occurrences of a letter a in w. By $w^{\mathrm{R}} = a_n a_{n-1} \dots a_1$ we refer to the *reversal* of the word w. Let $\cdot : \Delta^* \times \Delta^* \to \Delta^*$ defined by $u \cdot v = uv$ be the *concatenation* of words. Clearly, $\cdot$ is an associative operation and we have $u \cdot \varepsilon = u = \varepsilon \cdot u$. Therefore, ε forms the unit element. Consequently, $\Delta^* = (\Delta^*, \cdot, \varepsilon)$ is a monoid. Let $\text{factors}(w) = \{ w[i:j] \mid i, j \in [n]_0, i \leqslant j \}$ denote the set of all *factors* of w. Moreover, let $\text{prefix}(w) = \{ w[:j] \mid j \in [n] \}$ respectively $\text{suffix}(w) = \{ w[i:] \mid i \in [n] \}$ denote the set of all *prefixes* respectively *suffixes* of w. Let Δ' be an alphabet. A mapping $h : \Delta^* \to \Delta'^*$ satisfying $h(uv) = h(u)h(v)$ for all $u, v \in \Delta^*$ is called a *morphism*. In particular, for a morphism h we have $h(\varepsilon) = \varepsilon$ and by defining h for each $a \in \Delta$ the mapping is completely specified.

We call the function $\mathsf{parikh} : \Delta^* \times \Delta \to \mathbb{N}$ defined by $\mathsf{parikh}(w, a) = |w|_a$ the *Parikh vector* for $w \in \Delta^*$ and $a \in \Delta$.

A *finite automaton* is a structure $M = (Q, \Delta, \delta, q_0, F)$ where Q is the set of states, Δ an alphabet, $\delta : Q \times \Delta \to 2^Q$ a transition function, $q_0 \in Q$ the initial state, and $F \subseteq Q$ a set of accepting states. We call M a *deterministic finite automaton* (DFA) if for all $q \in Q$ and $a \in \Delta$ we have $(q, a) \in \mathsf{dom}(\delta)$ and $|\delta(q, a)| = 1$. Otherwise, M is a *non-deterministic finite automaton* (NFA). We say M *accepts* a word $w \in \Delta$ if there is a path via δ leading from q_0 to some $f \in F$ via edges labelled by $w[i]$ for each $i \in [|w|]_0$ (shortly $w \in L(M)$).

Let $M = (Q, \Delta, \delta, q_0, F)$ be an NFA. Whenever $|\Delta| = |\{ a \}| = 1$, we call M a *unary automaton*. Without loss of generality, we assume $q_0 \notin \delta(q, a)$ for all $q \in Q$. The automaton M is in *Chrobak normal form* [34] if there exists a path $q_0, \dots, q_n \in Q$ such that $\delta(q_i, a) = \{ q_{i+1} \}$ for all $i \in [n-1]$ and $n \in \mathcal{O}\left(|Q|^2\right)$. Secondly, M contains $\ell \leqslant |Q|$ cycles $q^j_{j_0}, \dots, q^j_{j_{m_j}}$ for $j \in [\ell]$ and $m_j \in \mathbb{N}$ such that $\delta(q^j_{j_i}, a) = \left\{ q^j_{j_i+1} \right\}$ for $i \in [m-2]$, $\delta(q^j_{j_{m_j}}, a) = \left\{ q^j_{j_0} \right\}$,

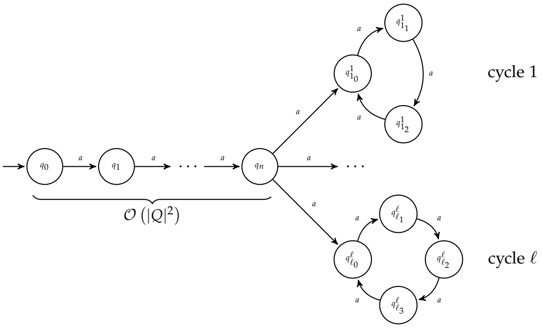


Figure 2.1. Automaton in Chrobak normal form

and $q^j_{j_0} \in \delta(q_n, a)$. Additionally, we have $\bigcap_{j\in[\ell]} \left\{ q^j_{j_i} \;\middle|\; i \in [m_j] \right\} = \varnothing$ and $q \notin \bigcup_{j\in[\ell]} \left\{ q^j_{j_i} \;\middle|\; i \in [m_j] \right\}$ for $q \in \{ q_0, \ldots, q_n \}$, as well as $\sum_{j\in[\ell]} \left(m_j + 1\right) \leqslant |Q|$. Such an automaton is given in Figure 2.1.

We shall generally distinguish between two alphabets, namely a finite set $A = \{ a, b, c, \ldots \}$ called *terminals* or *constants* and a possibly infinite set $\mathcal{X} = \{ \mathsf{x}_1, \mathsf{x}_2, \ldots \}$ called *variables* such that $A \cap \mathcal{X} = \varnothing$. We call a word $\alpha \in \mathtt{Pat}_A = (A \cup \mathcal{X})^*$ a *pattern*.

2.1 Logics

Within this section we follow the outline made in [46]. A finite nonempty set consiting of *relations* – anotated with their arity e.g. R/n for an n-ary relation R, *functions* – annotated with their arity e.g $f//n$ for an n-ary function f, and *constants* are called *vocabularies*. On top of a vocabulary $\mathcal{V}$ let $\mathcal{A}$ be a *(single-sorted) structure* (for short $\mathcal{V}$-structure) consisting of a nonempty set A, called *domain* of $\mathcal{A}$, an n-ary relation R^A for every n-ary relation $R/n \in \mathcal{V}$, an n-ary function f^A for every n-ary function $f//n \in \mathcal{V}$ and an element c^A for each constant $c \in \mathcal{V}$. For ease of readability we use infix and prefix notions of functions and relations interchangeably.

In various cases we are interested in expressing claims using constants, functions and relations ranging over arbitrarily but finitely many different domains $A_1, \ldots, A_n$ for $n \in \mathbb{N} \setminus \{ 0 \}$. To this extend we introduce *many-sorted* structures $\mathcal{A}$ for a vocabulary $\mathcal{V}$ (for short many-sorted $\mathcal{V}$-structure) where our structure $\mathcal{A}$ might contain more than one domain. To grasp the types within a many-sorted structure $\mathcal{A}$ holding domains $A_1, \ldots, A_n$ for $n \in \mathbb{N} \setminus \{ 0 \}$, we define analogously to the single-sorted $\mathcal{V}$-structure an m-ary relation $R^{A_{i_1} \cdots A_{i_m}}$ for $i_j, j \in [n]_0$ for every m-ary relation $R/m \in \mathcal{V}$, an k-ary function $f^{A_{i_1} \ldots A_{i_k} \to A_\ell}$ for $i_j, j, \ell \in [n]_0$ for every k-ary function $f//k \in \mathcal{V}$ and an element c^{A_ℓ} for at least one $\ell \in [n]_0$ and each constant $c \in \mathcal{V}$ for $m, k \in \mathbb{N}$. Whenever we consider an m-ary relation $R^{A_{i_1} \cdots A_{i_m}}$ such that $A_{i_1} = \ldots = A_{i_m}$ holds, we simply – as for the single-sorted structures – write $R^{A_{i_1}}$. Consequently, for an k-ary function $f^{A_{i_1} \ldots A_{i_k} \to A_\ell}$ whenever $A_{i_1} = \ldots = A_{i_k} = A_\ell$ we write $f^{A_{i_1}}$.

2.1.1 Propositional logic

Let $\textsc{Pl} = \{ \dot{0}, \dot{1} \}$ be a vocabulary purely holding two constants. We define the set of *propositional logic formulae* of the vocabulary $\textsc{Pl}$, namely PL, inductively as follows: 1. $\mathsf{0}, \mathsf{1} \in \mathsf{PL}$, 2. $\mathsf{x} \in \mathsf{PL}$ for each $\mathsf{x} \in \mathcal{X}$, 3. $\neg\varphi \in \mathsf{PL}$ for a formula $\varphi \in \mathsf{PL}$, and 4. $\varphi \vee \psi \in \mathsf{PL}$ for formulae $\varphi, \psi \in \mathsf{PL}$. Note, that the commonly known connectives can be derived by the above definition. For formulae $\varphi, \psi \in \mathsf{PL}$ let $\varphi \wedge \psi$, $\varphi \to \psi$, and $\varphi \leftrightarrow \psi$ be abbreviations for $\neg(\neg\varphi \vee \neg\psi)$, $\neg\varphi \vee \psi$, and $(\varphi \to \psi) \wedge (\psi \to \varphi)$, respectively. Let $\texttt{vars} : \mathsf{PL} \to 2^{\mathcal{X}}$ denote all variables within a propositional logic formula, defined inductively as follows: 1. $\texttt{vars}(\mathsf{0}) = \texttt{vars}(\mathsf{1}) = \emptyset$, 2. $\texttt{vars}(\mathsf{x}) = \{ x \}$ for all $x \in \mathcal{X}$, 3. $\texttt{vars}(\neg\varphi) = \texttt{vars}(\varphi)$ for all $\varphi \in \mathsf{PL}$, and 4. $\texttt{vars}(\varphi \vee \psi) = \texttt{vars}(\varphi) \cup \texttt{vars}(\psi)$ for all $\varphi, \psi \in \mathsf{PL}$.

We will now introduce the semantics of propositional logic formulae PL. To achieve this, we define a *satisfaction relation* $\models$ for a formula $\varphi \in \mathsf{PL}$. Let $\mathbb{B} = \{ \mathsf{0}, \mathsf{1} \}$ be a set and $\mathcal{B} = \left\{ \mathbb{B}, \dot{0}^{\mathbb{B}}, \dot{1}^{\mathbb{B}} \right\}$ be a PL-structure such that $\dot{0}^{\mathbb{B}} = \mathsf{0}$ and $\dot{1}^{\mathbb{B}} = \mathsf{1}$. An *assignment* $h : A \cup \mathcal{X} \to A$ is a morphism such that $h(\mathsf{x}) \in \mathbb{B}$ and $h(b) = b$ for $b^{\mathbb{B}} \in \left\{ \dot{0}^{\mathbb{B}}, \dot{1}^{\mathbb{B}} \right\}$ holds. To extend and modify a partial assignment we define for a variable $\mathsf{x} \in \mathcal{X}$ and $b \in \mathbb{B}$ the notation

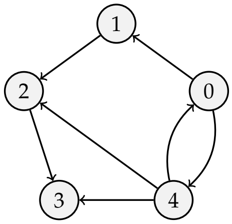


Figure 2.2. Directed Graph G as an example to model a propositional logic formula

$h\left[\frac{\mathsf{x}}{b}\right] = \{\, \mathsf{x}' \mapsto h(\mathsf{x}') \mid \mathsf{x}' \in \mathsf{dom}(h)\setminus\{\,\mathsf{x}\,\}\,\} \cup \{\, \mathsf{x} \mapsto b^{\mathbb{B}} \,\}$. Let

$$\mathcal{H}_{\mathcal{B}} = \left\{ h \,\middle|\, h : A \cup \mathcal{X} \to A \text{ morphism}, \forall\, b \in \mathbb{B} : h(b) = b^{A} \right\}$$

denote the set of all assignments.

Let $h \in \mathcal{H}_{\mathcal{B}}$ be an assignment. We define $h \models \varphi$ inductively for variables $\mathsf{x} \in \mathcal{X}$, constants $b \in \mathbb{B}$, and formulae $\varphi, \psi \in \mathsf{PL}$ by

$$\begin{aligned} h \models b & \;\; \textit{iff} \;\; b = 1, \\ h \models \mathsf{x} & \;\; \textit{iff} \;\; h(\mathsf{x}) = 1, \\ h \models \neg\varphi & \;\; \textit{iff} \;\; \textit{not } h \models \varphi, \text{and} \\ h \models \varphi \vee \psi & \;\; \textit{iff} \;\; h \models \varphi \text{ or } h \models \psi. \end{aligned}$$

We call a propositional logic formula *satisfiable* if there exists an assignment $h \in \mathcal{H}_{\mathcal{B}}$ such that $h \models \varphi$ holds. Consequently, we call φ *unsatisfiable* if there does not exist an assignment $h \in \mathcal{H}_{\mathcal{B}}$ such that $h \models \varphi$ holds and shortly write $\not\models \varphi$.

Example 2.1. As an example how to use propositional logic in practice, we model a reachability question within the directed graph G given in Figure 2.2. We can model this graph by the propositional logic formula

$$\begin{aligned} \varphi_G = \quad & \mathsf{x}_0 \to \mathsf{x}_1 \wedge \mathsf{x}_0 \to \mathsf{x}_4 \wedge \mathsf{x}_1 \to \mathsf{x}_2 \wedge \mathsf{x}_2 \to \mathsf{x}_3 \\ & \wedge \mathsf{x}_4 \to \mathsf{x}_0 \wedge \mathsf{x}_4 \to \mathsf{x}_2 \wedge \mathsf{x}_4 \to \mathsf{x}_3, \end{aligned}$$

using propositional logic variable $\mathsf{x}_i \in \mathcal{X}$ for each node $i \in [4]$. The formula φ_G models each edge of the graph G as a logical consequence. This allows moving between the nodes.

We now ask ourselves whether we can reach node 3 starting at node 0. To do so we simply ask whether any assignment satisfying the formula

$$(\varphi_G \wedge \mathsf{x}_0)$$

also satisfies x_3 which is indeed fulfilled. Consequently, we have

$$(\varphi_G \wedge \mathsf{x}_0) \models \mathsf{x}_3$$

and therefore we can find a path from node 0 to node 3.

2.1.2 First order logic

Let $\mathcal{V}$ be a vocabulary. A *term* of a vocabulary is a variable within our set $\mathcal{X}$, a constant in $\mathcal{V}$ or an n-ary function $f//n \in \mathcal{V}$, shortly called $\mathcal{V}$*term*. We define the set of *first order logic formulae* of the vocabulary $\mathcal{V}$, namely $\mathsf{FO}(\mathcal{V})$, inductively as follows: 1. $t_0 \doteq t_1 \in \mathsf{FO}(\mathcal{V})$ for $\mathcal{V}$ terms t_0 and t_1, 2. $R(t_0, \ldots, t_n) \in \mathsf{FO}(\mathcal{V})$ for $\mathcal{V}$ terms $t_0, \ldots, t_n \in \mathcal{V}$ and relation $R/n \in \mathcal{V}$ for $n \in \mathbb{N}$, 3. $\neg\varphi \in \mathsf{FO}(\mathcal{V})$ for a formula $\varphi \in \mathsf{FO}(\mathcal{V})$, 4. $\varphi \vee \psi \in \mathsf{FO}(\mathcal{V})$ for formulae $\varphi, \psi \in \mathsf{FO}(\mathcal{V})$, and 5. $\exists \mathsf{x}\,.\,\varphi$ for a formula $\varphi \in \mathsf{FO}(\mathcal{V})$. All formulae obtained by the inductive base cases 1. and 2. are called *atomic formulae*. Again, as stated in the previous section for propositional logic, we can derive derive the commonly known operations by the above definition. For formulae $\varphi, \psi \in \mathsf{FO}(\mathcal{V})$ let $\varphi \wedge \psi$, $\varphi \rightarrow \psi$, $\varphi \leftrightarrow \psi$, and $\forall \mathsf{x}\,.\,\varphi$ be abbreviations for $\neg(\neg\varphi \vee \neg\psi)$, $\neg\varphi \vee \psi$, $(\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$, and $\neg\exists \mathsf{x}\,.\,\varphi$, respectively. Let $\mathtt{vars}\,(\varphi)$, as for propositional logic formulae, denote all variables within a formula $\varphi \in \mathsf{FO}(\mathcal{V})$. By $\mathtt{bounded}\,(\varphi)$ we refer to all *bounded variables* occurring in φ being bound by a quantifier $Q \in \{\,\exists, \forall\,\}$, formally defined as follows:

$$\mathtt{bounded}\,(\varphi) = \begin{cases} \emptyset & \text{if } \varphi \text{ is a } \mathcal{V} \text{ term,} \\ \mathtt{bounded}\,(\psi) & \text{if } \varphi = \neg\psi, \\ \mathtt{bounded}\,(\psi_1) \cup \mathtt{bounded}\,(\psi_2) & \text{if } \varphi = \psi_1 \vee \psi_2, \\ \{\,\mathsf{x}\,\} \cup \mathtt{bounded}\,(\psi) & \text{if } \varphi = \exists \mathsf{x}\,.\,\psi. \end{cases}$$

By $\mathtt{free}\,(\varphi) = \mathtt{vars}\,(\Phi) \backslash \mathtt{bounded}\,(\varphi)$ we denote the *free variables* within φ. Furthermore, we denote the set of all atoms in φ by $\mathtt{atoms}\,(\varphi)$

We will now introduce the semantics of first order logic formulae $\mathsf{FO}(\mathcal{V})$ over our vocabulary $\mathcal{V}$. To achieve this, we define a *satisfaction relation* $\models$ for a $\mathcal{V}$-structure $\mathcal{A}$ and a formula $\varphi \in \mathsf{FO}(\mathcal{V})$. Let $\mathcal{A}$ be a $\mathcal{V}$-structure having the domain A. An *assignment* $h : A \cup \mathcal{X} \to A$ is a morphism such that $h(\mathsf{x}) \in A$ and $h(\mathsf{c}) = c^A$ holds. To extend and modify a partial assignment we define for a variable $\mathsf{x} \in \mathcal{X}$ and $c \in A$ the notation $h\left[\frac{\mathsf{x}}{c}\right] = \{\, \mathsf{x}' \mapsto h(\mathsf{x}') \mid \mathsf{x}' \in \mathsf{dom}(h) \backslash \{\, \mathsf{x} \,\} \,\} \cup \{\, \mathsf{x} \mapsto c^A \,\}$. Let

$$\mathcal{H}_\mathcal{A} = \left\{\, h \;\middle|\; h : A \cup \mathcal{X} \to A \text{ morphism}, \forall\, c \in A : h(c) = c^A \,\right\}$$

denote the set of all assignments.

Let $h \in \mathcal{H}_\mathcal{A}$ be an assignment. We define $\mathcal{A}, h \models \varphi$ inductively for terms $t_0, \ldots, t_n, R/n \in \mathcal{V}$ for $n \in \mathbb{N}$ and formulae $\varphi, \psi \in \mathsf{FO}(\mathcal{V})$, by

$$\begin{aligned}
\mathcal{A}, h \models t_0 \doteq t_1 & \quad \textit{iff} \;\; h(t_0) = h(t_1), \\
\mathcal{A}, h \models R(t_0, \ldots, t_n) & \quad \textit{iff} \;\; R^A(h(t_0), \ldots, h(t_n)), \\
\mathcal{A}, h \models \neg\varphi & \quad \textit{iff} \;\; \textit{not } \mathcal{A}, h \models \varphi \text{ (for short } \mathcal{A}, h \not\models \varphi), \\
\mathcal{A}, h \models \varphi \vee \psi & \quad \textit{iff} \;\; \mathcal{A}, h \models \varphi \text{ or } \mathcal{A}, h \models \psi, \text{and} \\
\mathcal{A}, h \models \exists\, \mathsf{x} \,.\, \varphi & \quad \textit{iff} \;\; \textit{there exists an } c \in A \textit{ such that} \mathcal{A}, h \left[\frac{\mathsf{x}}{c}\right] \models \varphi.
\end{aligned}$$

We call a $\mathcal{V}$ formula in a $\mathcal{V}$-structure $\mathcal{A}$ *satisfiable* if there exists an assignment $h \in \mathcal{H}_\mathcal{A}$ such that $\mathcal{A}, h \models \varphi$ holds and use $\mathcal{A} \models \varphi$ as a short form. In this case we also call h a *solution* of φ. Consequently, we call φ *unsatisfiable* if there does not exist an assignment $h \in \mathcal{H}_\mathcal{A}$ such that $\mathcal{A}, h \models \varphi$ holds and shortly write $\mathcal{A} \not\models \varphi$. A set $\Phi \subseteq \mathsf{FO}(\mathcal{V})$ of $\mathcal{V}$ formulae is satisfiable within a $\mathcal{V}$-structure $\mathcal{A}$ if there exists an assignment $h \in \mathcal{H}_\mathcal{A}$ such that $\mathcal{A}, h \models \varphi$ holds for all $\varphi \in \Phi$ and we denote this by $\mathcal{A} \models \Phi$. Otherwise, the set of formulae Φ is unsatisfiable within the $\mathcal{V}$-structure $\mathcal{A}$ ($\mathcal{A} \not\models \Phi$). Continuing this trail, a formula φ is a *consequence* of Φ ($\mathcal{A}, \Phi \models \varphi$) if whenever $\mathcal{A} \models \Phi$ we have $\mathcal{A} \models \varphi$. If φ is true in all structures under all assignments we call it *valid* ($\models \varphi$).

For the remainder of this work let $\top$ denote a constant such that $\models \top$ and $\bot$ denote a constant such that $\not\models \bot$, meaning $\top$ is a formula which is always valid and $\bot$ is always unsatisfiable.

Remark 2.2. Within this work we usually omit restating the $\mathcal{V}$-structure $\mathcal{A}$ whenever it is clear from the context. Therefore for a $\mathcal{V}$ formula $\varphi \in$

$\mathsf{FO}(\mathcal{V})$ and an assignment $h \in \mathcal{H}_{\mathcal{A}}$ instead of $\mathcal{A}, h \models \varphi$ we write $h \models \varphi$. Consequently, we omit the $\mathcal{V}$-structure $\mathcal{A}$ whenever talking about the general satisfiability of a formula and write $\models \varphi$ instead of $\mathcal{A} \models \varphi$.

Whenever the connection of constant c^A, functions f^A or relation R^A to its $\mathcal{V}$-structure is clear from context we omit the superscript ^{A} and simply write c, f, and R, instead of c^A, f^A, and R^A, respectively.

There exist several normal forms for first order logic formulae. A form which plays an important role in specifying the fragments of formulae is the *prenex normal form*, that is a formula $Q_1 \, \mathsf{x}_1 \,.\, Q_2 \, \mathsf{x}_2 \,.\, \ldots Q_n \, \mathsf{x}_n \,.\, \psi$ where $Q_i \in \{ \exists, \forall \}$ and $n \in \mathbb{N}$, $i \in [n]$ and ψ does not contain any quantifiers. It can be shown that for any first order logic formula, we are able to gather an equivalent formula in prenex normal form. Based on formulae $\varphi = Q_1 \, \mathsf{x}_1 \,.\, Q_2 \, \mathsf{x}_2 \,.\, \ldots Q_n \, \mathsf{x}_n \,.\, \psi$ in prenex normal form we define the Σ_k fragment for $k \in [n]$. φ is part of the Σ_k fragment if it contains at most k consecutive quantifier blocks where each quantifier block consists of either existential or universal quantifiers, adjacent block contain different quantifiers, and the initial block holds existential quantifiers, e.g. $\exists \, \mathsf{x}_1 \,.\, \exists \, \mathsf{x}_2 \,.\, \forall \, \mathsf{x}_3 \,.\, \exists \, \mathsf{x}_4 \,.\, \psi$ where ψ is quantifier free is part of the Σ_3 fragment.

Example 2.3. Let $\mathcal{V} = \{ E/2 \}$ be the vocabulary of a graph containing the 2-ary relation E. The $\mathcal{V}$-structure $\mathcal{G} = (V, E^V)$ represents undirected graphs, whenever 1. for all $v \in V$ it does not hold $E^V(v, v)$, and 2. for all $v_1, v_2 \in V$ if $E^V(v_1, v_2)$ we have $E^V(v_2, v_1)$. These conditions form our axioms of undirected graphs and lead to the formalisation 1. $\forall \, \mathsf{x}_1 \,.\, E(\mathsf{x}_1, \mathsf{x}_1)$, and 2. $\forall \, \mathsf{x}_1 \,.\, \forall \, \mathsf{x}_2 \,.\, (E(\mathsf{x}_1, \mathsf{x}_2) \rightarrow \neg E(\mathsf{x}_2, \mathsf{x}_1))$ within $\mathsf{FO}(\mathcal{V})$.

Considering the above given $\mathcal{V}$-structure $\mathcal{G} = (V, E^V)$ such that $V = \{ 0, 1, 2, 3, 4 \}$ and

$$\begin{aligned} E^V = \{&(0,1),(1,0),(4,0),(0,4),(1,2),(2,1),(2,3),\\ &(3,2),(4,2),(2,4),(4,3),(3,4)\} \end{aligned}$$

– an undirected version of the graph seen in Example 2.1 and an assignment $h = \{ \mathsf{x}_1 \mapsto 0, \mathsf{x}_2 \mapsto 3 \}$ we have $\mathcal{G}, h \models \exists \, \mathsf{x}_3 \,.\, E(\mathsf{x}_1, \mathsf{x}_3) \wedge E(\mathsf{x}_3, \mathsf{x}_2)$, meaning we are asking whether there exists a path from node 0 to node 3 by traversing one node.

2.2 Logical Theories

Within this work the basis is formed by three different logical theories, namely word equations, linear inequalties (also called linear length constraints), and regular membership constraint. This section introduces these theories as instances of first order logic, meaning we introduce a vocabulary and an axiomatisation for each theory and instantiate it by defining an appropriate structure.

2.2.1 Word equations

Given the vocabulary $\mathcal{W} = \{\, \cdot /\!/ 2, \dot{\varepsilon} \,\}$. For ease of readability we write $\cdot$ as infix operation. We consider the Σ_1 fragment of the first order logic formulae $\mathsf{FO}(\mathcal{W})$ having the axioms 1. $\forall \mathrm{x}_1 \,.\, \forall \mathrm{x}_2 \,.\, \forall \mathrm{x}_3 \,.\, (x_1 \cdot x_2) \cdot x_3 \doteq x_1 \cdot (x_2 \cdot x_3)$, and 2. $\forall \mathrm{x}_1 \,.\, \dot{\varepsilon} \cdot x_1 \doteq x_1 \wedge x_1 \cdot \dot{\varepsilon} \doteq x_1$, that are the associativity of $\cdot /\!/ 2$ and the existence of a neutral element. In other words we expect our structure to form a monoid. Consider the $\mathcal{W}$-structure $\mathcal{A}^{\doteq} = \{\, A^*, \cdot^A, \dot{\varepsilon}^A \,\}$, where $\cdot^A$ is defined as the concatenation of words seen in the previous section. Clearly, this structure meets our axioms choosing $\dot{\varepsilon}^A = \varepsilon$ as the neutral element. In generally within the first order logic theory we are interested in checking whether we can unify two pattern to make them equal in terms of our alphabet A. The only atomic formula is a word equation formally given as follows:

Definition 2.4. Let $\alpha, \beta \in \mathtt{Pat}_A$ be $\mathcal{W}$ terms. The formula $\alpha \doteq \beta$ is called a *word equation*. Furthermore, we call a set of word equations E a *system of word equations*.

By left hand side respectively right hand side (or short sides) of the word equation $\alpha \doteq \beta$ we refer to the pattern α, respectively β. Let $\mathsf{WEQ}_A = \mathtt{Pat}_A \times \mathtt{Pat}_A$ denote the set of all word equations over $\mathtt{Pat}_A$. The length of a word equation is a natural extension of the length function defined for words. For a word equation $\alpha \doteq \beta$ the length is given by $|\alpha| + |\beta|$ and the length of a system of word equations E is given by $\sum_{\alpha \doteq \beta \in E} (|\alpha| + |\beta|)$. The length of a solution h of a WE $\alpha \doteq \beta$ is given by $2 \cdot |h(\alpha)|$ which naturally extends to system of word equations. We call a solution h *minimal* whenever there does not exist another solution with shorter length. Recall, in this section we only consider the $\mathcal{W}$-structure $\mathcal{A}^{\doteq}$ and therefore omit $\mathcal{A}^{\doteq}$ while using the satisfiability relation $\models$.

Example 2.5. Consider the system of word equations

$$E = \{\, \mathsf{x}_1 ab \mathsf{x}_2 \doteq a\mathsf{x}_1\mathsf{x}_2 b, \mathsf{x}_1 b \doteq a\mathsf{x}_2 \,\}$$

and substitution $h = \{\, \mathsf{x}_1 \mapsto a, \mathsf{x}_2 \mapsto b \,\}$. By applying the substitution to the system of word equations we get

$$h\,(\mathsf{x}_1 ab \mathsf{x}_2) = h\,(\mathsf{x}_1)\ ab\, h\,(\mathsf{x}_2) = aabb = a\, h(\mathsf{x}_1)\, h(\mathsf{x}_2)\, b = h(a\mathsf{x}_1\mathsf{x}_2 b)$$

and

$$h(\mathsf{x}_1 b) = ab = h(a\mathsf{x}_2).$$

Therefore $h \models E$, since h is a solution to both equations.

Karhumäki, Mignosi, and Plandowski [68] showed that for every system of word equations, a single equation can be constructed which is satisfiable if and only if the initial system was satisfiable. The solution to the constructed word equation can be directly transferred to a solution of the original word equation system. Therefore, systems of word equations do not add expressiveness. The following theorem states this results.

Theorem 2.6. *[68] Let $e_1 = \alpha_1 \doteq \beta_1, e_2 = \alpha_2 \doteq \beta_2 \in \mathsf{WEQ}_A$ and let $a, b \in A$ such that $|e_1|_a = |e_2|_a = |e_1|_b = |e_2|_b = 0$ then we have*

$$h \models e_1 \wedge h \models e_2 \textit{ iff } h \models \alpha_1\, a\, \alpha_2 \alpha_1\, b\, \alpha_2 \doteq \beta_1\, a\, \beta_2 \beta_1\, b\, \beta_2.$$

The satisfiability problem for word equations is an important problem in both mathematics and computer science. For instance, in an attempt to solve Hilbert's tenth problem [61] in the negative, Markov showed a reduction from word equations to Diophantine equations (see [70, 85, 87]), in the hope that word equations would prove to be undecidable. However, Makanin [87] proved in 1977 that the satisfiability of word equations is, in fact, decidable. Considerable effort has also been spent identifying the complexity of deciding the satisfiability of a given equation. After a sequence of intermediate results [85], Plandowski [94] showed that this problem is in PSPACE. In a series of recent papers [65, 66], Jeż applied a new elegant technique called recompression to word equations to show that word equations can be solved in non-deterministic linear space. However, there is a mismatch between the aforementioned upper bounds and the only known lower bound: solving word equations is NP-hard.

2.2.2 Linear Diophantine inequalities

Given the vocabulary $\mathcal{L} = \{ +//2, \leqslant/2, \dot{0} \}$. We consider the Σ_1 fragment of the first order logic formulae $\mathsf{FO}(\mathcal{L})$ being characterised by the following axioms 1. $\forall \mathsf{x}_1 . \forall \mathsf{x}_2 . \forall \mathsf{x}_3 . (x_1 + x_2) + x_3 \doteq x_1 + (x_2 + x_3)$, 2. $\forall \mathsf{x}_1 . \forall \mathsf{x}_2 . \mathsf{x}_1 + x_2 \doteq x_2 + x_1$, and 3. $\forall \mathsf{x}_1 . \dot{0} + x_1 \doteq x_1 \wedge x_1 + \dot{0} \doteq x_1$, that are the associativity and commutativity of $+//2$ and the existence of a neutral element. Secondly, we are axiomatising $\leqslant/2$ to be an total ordering on our domain by adding 1. $\forall \mathsf{x}_1 . \mathsf{x}_1 \leqslant \mathsf{x}_1$, 2. $\forall \mathsf{x}_1 . \forall \mathsf{x}_2 . \mathsf{x}_1 \leqslant x_2 \vee x_2 \leqslant x_1$, and 3. $\forall \mathsf{x}_1 . \forall \mathsf{x}_2 . \forall \mathsf{x}_3 . (x_1 \leqslant x_2 \wedge x_2 \leqslant x_3) \to x_1 \leqslant x_3$, that are reflexivity, symmetry and transitivity of $\leqslant$. Moreover, we have monotonicity, that is $\forall \mathsf{x}_1 . \forall \mathsf{x}_2 . \forall \mathsf{x}_3 . \mathsf{x}_1 \leqslant \mathsf{x}_2 \to \mathsf{x}_1 + \mathsf{x}_3 \leqslant \mathsf{x}_2 + \mathsf{x}_3$. Consider the $\mathcal{L}$-structure $\mathcal{A}_l = \{ \mathbb{Z}, +^{\mathbb{Z}}, \leqslant^{\mathbb{Z}}, \dot{0}^{\mathbb{Z}} \}$, where $+^{\mathbb{Z}}$ and $\leqslant^{\mathbb{Z}}$ are defined as the commonly known addition respectively total ordering within $\mathbb{Z}$. Clearly, this structure meets our axioms choosing $\dot{0}^{\mathbb{Z}} = 0$ as the neutral element. As an abbreviation for repeated addition we introduce the multiplication by a constant using $\cdot$, e.g. we write $\mathsf{x}_1 + \mathsf{x}_1 + \mathsf{x}_1$ as $3 \cdot \mathsf{x}_1$. In general we are interested in finding positive integer solutions to linear inequalities – commonly called linear Diophantine inequalities which meet the two axioms of $\mathsf{FO}(\mathcal{L})$ and are defined as following:

Definition 2.7. A linear Diophantine (in)equality is defined by

$$\sum_{\mathsf{x} \in \mathcal{X}} c_{\mathsf{x}} \cdot \mathsf{x} \bowtie c$$

where $c, c_{\mathsf{x}} \in \mathbb{Z}$, $\bowtie \; \in \{ \leqslant, \doteq \}$ and $\mathsf{x} \in \mathcal{X}$ are variables. We denote the set of all linear (in)equalities by LinC.

Note, the commonly known orderings can be derived by using $\leqslant$ of our vocabulary $\mathcal{L}$. Using the rules described in [6] allows us to use the orderings $<, >, \geqslant$ as abbreviations as following:

$$\sum_{\mathsf{x} \in \mathcal{X}} c_{\mathsf{x}} \cdot \mathsf{x} \doteq c \text{ reduces to } \sum_{\mathsf{x} \in \mathcal{X}} c_{\mathsf{x}} \cdot \mathsf{x} \leqslant c \text{ and } \sum_{\mathsf{x} \in \mathcal{X}} c_{\mathsf{x}} \cdot \mathsf{x} \geqslant c,$$

$$\sum_{\mathsf{x} \in \mathcal{X}} c_{\mathsf{x}} \cdot \mathsf{x} < c \text{ reduces to } \sum_{\mathsf{x} \in \mathcal{X}} c_{\mathsf{x}} \cdot \mathsf{x} \leqslant c + (-1),$$

$$\sum_{\mathsf{x}\in\mathcal{X}} c_{\mathsf{x}} \cdot \mathsf{x} \geqslant c \text{ reduces to } \sum_{\mathsf{x}\in\mathcal{X}} -c_{\mathsf{x}} \cdot \mathsf{x} \leqslant -c,$$
$$\sum_{\mathsf{x}\in\mathcal{X}} c_{\mathsf{x}} \cdot \mathsf{x} > c \text{ reduces to } \sum_{\mathsf{x}\in\mathcal{X}} -c_{\mathsf{x}} \cdot \mathsf{x} \leqslant -c + (-1).$$

Nevertheless, for the remainder of this work we restrict our setting, without loss of generality, to linear Diophantine inequalities with $\leqslant$ as the only operator.

Example 2.8. Consider the system of linear inequalities

$$L = \{\, \mathsf{x}_1 + 2 \cdot \mathsf{x}_2 \leqslant 3, \mathsf{x}_1 + 2 \leqslant 5 \,\}$$

and substitution $h = \{\, \mathsf{x}_1 \mapsto 2, \mathsf{x}_2 \mapsto 0 \,\}$. By applying the substitution to L we get

$$h\,(\mathsf{x}_1 + 2 \cdot \mathsf{x}_2) = 2 + 2 \cdot 0 = 2 \leqslant 3$$

and

$$h(\mathsf{x}_1 + 2) = 2 + 2 \leqslant 5.$$

Therefore $h \models L$, since h is a solution to both linear inequalities.

Finding solutions to linear inequalities can be divided into two questions: mapping all variables to 1. integer solutions over $\mathbb{Z}$, and 2. postive integer solutions over $\mathbb{N}$. Considering integer solutions is known to be solvable in deterministic polynomial time [36], whereas it was proven to be NP-complete due to a reduction based on the subset sum problem if we restrict ourselves to positive solutions (cf. [67] and the references therein).

2.2.3 Regular membership constraints

As for word equations regular membership constraints are built on top of two disjoint alphabets: terminals A and variables $\mathcal{X}$. In general a regular constraints asks whether a pattern $\alpha \in \mathtt{Pat}_A$ is a member of a regular language defined by a regular expression over $\mathtt{Pat}_A$.

Given the vocabulary $\mathcal{R}_e = \{\, \cdot//2, \cup//2, {}^*//1, {}^-//1, \dot{\in}\,/2, \dot{\emptyset}, \dot{\varepsilon} \,\}$. We consider the Σ_1 fragment of the first order logic formulae $\mathsf{FO}(\mathcal{R})$ being axiomatized as 1. associativity and the existence of a neutral element $\dot{\varepsilon}$ of $\cdot//2$, 2. associativity, commutativity, the existence of a neutral element $\dot{\emptyset}$ and idempotents $\forall\ \mathsf{x}_1 . \mathsf{x}_1 \cup \mathsf{x}_1 \doteq \mathsf{x}_1$ of $\cup//2$ and, 3. distributivity, $\forall \mathsf{x}_1 . \forall \mathsf{x}_2 . \forall \mathsf{x}_3 . \mathsf{x}_1 \cdot (\mathsf{x}_2 \cup \mathsf{x}_3) \doteq (\mathsf{x}_1 \cdot \mathsf{x}_2) \cup (\mathsf{x}_1 \cdot \mathsf{x}_3)$, 4. annihilation by $\dot{\varnothing}$, $\forall \mathtt{x}_1 . \mathtt{x}_1 \cdot \dot{\varnothing} \doteq \dot{\varnothing} \wedge \dot{\varnothing} \cdot \mathtt{x}_1 \doteq \dot{\varnothing}$, 5. $\forall \mathtt{x}_1 . \mathtt{x}_1 \leqslant \mathtt{x}_1$. In order to formally define our many-sorted $\mathcal{R}_e$-structure we define regular expressions $\mathtt{RegExC}_A$ over four operations, namely the *concatenation* $\cdot : \mathtt{RegExC}_A \times \mathtt{RegExC}_A \to \mathtt{RegExC}_A$, *union* $\cup : \mathtt{RegExC}_A \times \mathtt{RegExC}_A \to \mathtt{RegExC}_A$, *Kleene star* $^* : \mathtt{RegExC}_A \to \mathtt{RegExC}_A$, and *complement* $^- : \mathtt{RegExC}_A \to \mathtt{RegExC}_A$. On top of these operations we define the set of regular expressions $\mathtt{RegExC}_A$ inductively as follows: we have $\varepsilon, \varnothing, a \in \mathtt{RegExC}_A$ for $a \in A$. Furthermore, given $R_1, R_2 \in \mathtt{RegExC}_A$ we have $R_1 \cdot R_2, R_1 \cup R_2, R_1^*, \overline{R_1} \in \mathtt{RegExC}_A$.

Consider the many-sorted $\mathcal{R}_e$-structure

$$\mathcal{A}_e = \{ \mathtt{RegExC}_A, A^*, \cdot^A, \dot{\varepsilon}^A, \cdot^{\mathtt{RegExC}_A}, \cup^{\mathtt{RegExC}_A}, \\ *^{\mathtt{RegExC}_A}, -^{\mathtt{RegExC}_A}, \dot{\varnothing}^{\mathtt{RegExC}_A}, \dot{\varepsilon}^{\mathtt{RegExC}_A}, \dot{\in}^{A\,\mathtt{RegExC}_A} \},$$

where $\cdot^A$ is defined as the concatenation of words seen in the previous section and $\dot{\varepsilon}^A = \varepsilon \in A^*$. Our regular expressions operations over $\mathtt{RegExC}_A$ are defined as commonly known, as well as the constants $\dot{\varnothing}^{\mathtt{RegExC}_A}$ and $\dot{\varepsilon}^{\mathtt{RegExC}_A}$

The semantics $L : \mathtt{RegExC}_A \to 2^{\mathtt{Pat}}_A$ are given by $L(a) = \{ a \}$ for $a \in A \cup \{ \varepsilon \}$, $L(\varnothing) = \varnothing$. For $R_1, R_2 \in \mathtt{RegExC}_A$, let $R_1 \cdot R_2 = \{ \alpha \cdot \beta \mid \alpha \in L(R_1), \beta \in L(R_2) \}$, $R_1 \cup R_2 = L(R_1) \cup L(R_2)$, $L(R_1^*) = L(R_1)^*$, and $L(\overline{R_1}) = A^* \backslash L(R_1)$. Note, all regular expression terms must be grounded (i.e. cannot contain variables). The relation $\dot{\in}^{A\,\mathtt{RegExC}_A}$ is defined as the classical set membership predicate $\in$.

Clearly, this many-sorted structure meets our axioms, as they form a combination of the previously seen monoid and a classical Kleene-algebra, together with a new membership predicate $\dot{\in}^{A\,\mathtt{RegExC}_A}$.

We are now ready to define a regular membership constraint.

Definition 2.9. Let $\alpha \in \mathtt{Pat}_A$ and $R \in \mathtt{RegExC}_A$. We call

$$\alpha \,\dot{\in}^{A\,\mathtt{RegExC}_A}\, R$$

a *regular membership constraint*. Let Reg_A denote the set of all regular membership constraints.

Example 2.10. Consider the set of regular membership constraints

$$R = \{ \mathtt{x}_1 ab \mathtt{x}_2 \,\dot{\in}\, a(a \cup b)^*, \mathtt{x}_1 b \,\dot{\in}\, a^* b^* \}$$

and substitution $h = \{\, \mathsf{x}_1 \mapsto \varepsilon, \mathsf{x}_2 \mapsto b \,\}$. By applying the substitution to R we get $h\,(\mathsf{x}_1 ab \mathsf{x}_2) = h\,(\mathsf{x}_1)\; ab\, h\,(\mathsf{x}_2) = abb \mathrel{\dot{\in}} a(a \cup b)^*$ and $h(\mathsf{x}_1 b) = b \mathrel{\dot{\in}} a^* b^*$. Therefore $h \models R$, since h satisfies the set of regular membership constraints R.

Finding a solution within the theory of regular expression membership predicates is known to decidability [4]. Within this work, we reprove this result purely using classical automata-based methods in Section 4.3. There are many extensions for this theory influenced by practical needs, whereas for some of them we prove their decidability respectively undecidability in the aforementioned section.

2.3 Combination of logical theories

Typically the logical theories discussed in the previous section are combined to new many-sorted structures. Motivated by applications in formal verification, the expressiveness of the previously introduced theories alone is not sufficient. In the following we introduce a combination of linear inequalties with word equations and regular membership constraints as well as the combination of all aforementioned theories. Linear inequalties are introduced to reason about the length of a potential solution to a variable. Consequently, we introduce a new function mapping patterns to an integer. Let $\mathsf{len}^{A\to\mathbb{Z}}$ be the length function defined for a pattern $\alpha \in \mathtt{Pat}_A$ and an assignment $h \in \mathcal{H}_A$ by $\mathsf{len}^{A\to\mathbb{Z}}(\alpha) = |h(\alpha)|$. Linear constraints using the length function are commonly called *linear length constraints* and formally defined as follows:

Definition 2.11. A *linear length constraints* is defined by

$$\sum_{\mathsf{x}\in\mathcal{X}} c_{\mathsf{x}} \cdot \mathsf{len}(\mathsf{x}) \bowtie c$$

where $c, c_{\mathsf{x}} \in \mathbb{Z}$, $\bowtie\ \in \{\, \leqslant, \doteq \,\}$ and $\mathsf{x} \in \mathcal{X}$. We denote the set of all linear length constraints by Lin.

Note, a length constraint over an arbitrary pattern $\alpha \in \mathtt{Pat}_A$ can easily be brought into the above form by considering its *integer abstraction*, being defined by

$$\mathsf{abs}_A(\alpha) = \left(\sum_{\mathsf{x}\in\mathcal{X}} |\alpha|_{\mathsf{x}}\right) \cdot \mathsf{len}(\mathsf{x}) + \sum_{a\in A} |\alpha|_a.$$

In terms of decidability even though the theories themselves are decidable their combination might even be undecidable or of unknown status.

2.3.1 Word equations with length constraints

Consider the vocabulary

$$\mathcal{WL} = \mathcal{W} \cup \mathcal{L} \cup \{\, \mathsf{len}/\!/1 \,\} = \{\, \cdot/\!/2, \dot{\varepsilon}, +/\!/2, \leqslant /2, \dot{0}, \mathsf{len}/\!/1 \,\}.$$

We consider the Σ_1 fragment of the first order logic formulae $\mathsf{FO}(\mathcal{WL})$ having the axioms for the aforementioned fragments for word equations and linear Diophantine inequalties as introduced in Section 2.2.1 and 2.2.2. Note, the new function $\mathsf{len}/\!/1$ derives its axiom from the fragment of linear Diophantine inequalties. Consider the $\mathcal{WL}$-structure $\mathcal{A}_l^{\dot{=}} = \left\{\, A^*, \mathbb{Z}, \cdot^A, \dot{\varepsilon}^A, +^{\mathbb{Z}}, \leqslant^{\mathbb{Z}}, \dot{0}^{\mathbb{Z}}, \mathsf{len}^{A\to\mathbb{Z}} \,\right\}$, again simply being a combination of the two previously introduced fragments $\mathcal{A}^{\dot{=}}$ and $\mathcal{A}_l$, together with the function $\mathsf{len}^{A\to\mathbb{Z}}$. We denote the set all word equations with length constraints by $\mathsf{WL}_A = \mathsf{WEQ}_A \cup \mathsf{LinC} \cup \mathsf{Lin}$.

Example 2.12. We revisit the system of word equations

$$E = \{\, \mathsf{x}_1 ab \mathsf{x}_2 \doteq a \mathsf{x}_1 \mathsf{x}_2 b, \mathsf{x}_1 b \doteq a \mathsf{x}_2 \,\}$$

seen in Example 2.5 equipped with the set of linear length constraints

$$L = \{\, \mathsf{len}(\mathsf{x}_1) + 3 \leqslant \mathsf{len}(\mathsf{x}_2) \,\}.$$

The assignment $h = \{\, \mathsf{x}_1 \mapsto a, \mathsf{x}_2 \mapsto b \,\}$ does not yield a solution since $\mathsf{len}(h(\mathsf{x}_1)) + 3 = |a| + 3 = 4 \nleq 1 = |b| = \mathsf{len}(h(\mathsf{x}_2))$. In general the added length constraint disallows any solution to solely looking at the word equation. The word equation $\mathsf{x}_1 b \doteq a \mathsf{x}_2$ is only satisfiable whenever an assignment h satisfies $|h(\mathsf{x}_1)| = |h(\mathsf{x}_2)|$.

Prominently, the decidability of the theory of word equations enhanced with a length function remains a major open problem in theory. Catching up Markov's ideas, in [88, 89] a reduction from the more powerful theory of word equations with linear length constraints to Diophantine equations is shown.

2.3.2 Regular membership and length constraints

As for word equations and length constraints, the combination of regular expression membership constraints and length constraints over string variables is achieved by merging the two vocabularies $\mathcal{R}$ and $\mathcal{L}$, as well as adding the new length function. Let

$$\begin{aligned}\mathcal{RL} &= \mathcal{R}_e \cup \mathcal{L} \cup \{\, \mathsf{len}//1 \,\} \\ &= \{\, \cdot//2, \cup//2, {}^*//1, {}^-//1, \dot{\in}\ /2, \dot{\varnothing}, \dot{\varepsilon}, +//2, \leqslant\ /2, \dot{0}, \mathsf{len}//1 \,\}\end{aligned}$$

be a vocabulary. We consider the Σ_1 fragment of the first order logic formulae $\mathsf{FO}(\mathcal{RL})$ having the axioms introduced for regular membership constraints and linear Diophantine inequalties in Section 2.2.3 and 2.2.2. We define the many-sorted $\mathcal{RL}$-structure

$$\begin{aligned}\mathcal{A}_{el} = \{\, &\mathtt{RegExC}_A, A^*, \mathbb{Z} \cdot^A, \dot{\varepsilon}^A, \cdot^{\mathtt{RegExC}_A}, \cup^{\mathtt{RegExC}_A}, *^{\mathtt{RegEx}_A}, {}^{-\mathtt{RegExC}_A}, \\ &\dot{\varnothing}^{\mathtt{RegExC}_A}, \dot{\varepsilon}^{\mathtt{RegExC}_A}, \dot{\in}^{A\,\mathtt{RegExC}_A}, +^{\mathbb{Z}}, \leqslant^{\mathbb{Z}}, \dot{0}^{\mathbb{Z}}, \mathsf{len}^{A\to\mathbb{Z}} \,\},\end{aligned}$$

again simply being a combination of the two previously introduced fragments $\mathcal{A}_e$ and $\mathcal{A}_l$, together with the function $\mathsf{len}^{A\to\mathbb{Z}}$. We denote the set of all regular expression membership constraints with length constraints by $\mathsf{RL}_A = \mathsf{Reg}_A \cup \mathsf{LinC} \cup \mathsf{Lin}$.

Example 2.13. Consider the set of regular membership constraints seen in Example 2.10

$$R = \{\, \mathsf{x}_1 ab \mathsf{x}_2 \mathrel{\dot{\in}} a(a \cup b)^*, \mathsf{x}_1 b \mathrel{\dot{\in}} a^* b^* \,\}$$

and the linear length constraints

$$L = \{\, \mathsf{len}(\mathsf{x}_1) + 3 \leqslant \mathsf{len}(\mathsf{x}_2) \,\}\,.$$

The assignment $h = \{\, \mathsf{x}_1 \mapsto \varepsilon, \mathsf{x}_2 \mapsto aaa \,\}$ yields a solution since $h(\mathsf{x}_1 ab\ \mathsf{x}_2) = abaaa \mathrel{\dot{\in}} a(a \cup b)^*$, $h(\mathsf{x}_1 b) = b \mathrel{\dot{\in}} a^* b^*$ and $\mathsf{len}(\mathsf{x}_1) + 3 = \mathsf{len}(\varepsilon) + 3 = 3 \leqslant 3 = \mathsf{len}(aaa) = \mathsf{len}(\mathsf{x}_2)$. Therefore $h \models R \cup L$, since h satisfies the set of regular membership constraints R and the length constraints L.

The decidability of this combination was also proven in [4]. In Section 4.3 we reprove the decidability, again purely using automata-based methods. The proof for this theory will be used to design a decision procedure leading to an implementation of a solver for regular expression membership constraints involving linear length constraints over our variables.

2.3.3 Word equations combined with regular membership and length constraints

The largest and most expressive theory considered in this work is the combination of all aforementioned theories. In this sense the considered vocabulary $\mathcal{WRL}$ is the union of the theories combining word equations and length constraints and the combination of regular membership constraints and length constraints, meaning $\mathcal{WRL} = \mathcal{WL} \cup \mathcal{RL}$. Following this trail the logic formulae over $\mathsf{FO}(\mathcal{WRL})$ derive their axioms from the previously mentioned theories, too. Consequently, the many-sorted $\mathcal{WRL}$-structure $\mathcal{A}_{el}^{\doteq}$ is again obtained by the union of $\mathcal{A}_{l}^{\doteq}$ and $\mathcal{A}_{el}$. We denote the set of all so called *string constraints* by $\mathsf{WRL}_A = \mathsf{RL}_A \cup \mathsf{WL}_A$.

Example 2.14. Consider the word equation $a\mathsf{x}_1ab\mathsf{x}_2ab\mathsf{x}_1b \doteq \mathsf{x}_2ba\mathsf{x}_1aabb\mathsf{x}_1$, the length constraint $\mathsf{len}(\mathsf{x}_1) \cdot 2 \leqslant \mathsf{len}(\mathsf{x}_2)$, and the regular expression membership constraint $\mathsf{x}_1 \dot{\in} b^*$. The assignment $h = \{\, \mathsf{x}_1 \mapsto bb, \mathsf{x}_2 \mapsto abba \,\}$ is a solution to the conjunction of all constraints since we have

$$\begin{gathered} h(a\mathsf{x}_1ab\mathsf{x}_2ab\mathsf{x}_1b) = abbababbaabbbb = h(\mathsf{x}_2ba\mathsf{x}_1aabb\mathsf{x}_1), \\ \mathsf{len}(\mathsf{x}_1) \cdot 2 = |bb| \cdot 2 = 4 \leqslant 4 = |abba| = \mathsf{len}(\mathsf{x}_2), \text{ and} \\ h(\mathsf{x}_1) = bb \dot{\in} b^*. \end{gathered}$$

Solely considering the theory of word equations and regular expression membership predicates is known to be decidable [85]. We do not explicitly state this combination here, since it is of minor interest for the following chapters. However, it is not known whether the satisfiability problem for word equations involving all aforementioned theories is decidable or not. Moreover, already in the presence of other simple and natural constraints, like string-number conversion, which we will introduce in the next section, this problem becomes undecidable (cf. [41]).

Remark 2.15. To ease readability whenever we consider many-sorted the-

Table 2.1. Overview of the introduced theories.

	Vocabulary name	Vocabulary components	Structure name
Word equations	$\mathcal{W}$	$\{\cdot//2, \dot{\varepsilon}\}$	$\mathcal{A}^{\doteq}$
Linear Diophantine inequalities	$\mathcal{L}$	$\{+//2, \leqslant/2, \dot{0}\}$	$\mathcal{A}_l$
Regular membership constraints	$\mathcal{R}_e$	$\{\cdot//2, \cup//2, *//1, ^{-}//1, \dot{\in}/2, \dot{\emptyset}, \dot{\varepsilon}\}$	$\mathcal{A}_e$
Word equations with length constraints	$\mathcal{WL}$	$\mathcal{W} \cup \mathcal{L} \cup \{\mathsf{len}//1\}$	$\mathcal{A}_l^{\doteq}$
Regular membership and length constraints	$\mathcal{RL}$	$\mathcal{R}_e \cup \mathcal{L} \cup \{\mathsf{len}//1\}$	$\mathcal{A}_{el}$
Word equations with regular membership and length constraints	$\mathcal{WRL}$	$\mathcal{WL} \cup \mathcal{RL}$	$\mathcal{A}_{el}^{\doteq}$

ories and not only reason about variables within pattern over $\mathtt{Pat}_A$, we distinguish between *string* variables x and *integer* variables $\bar{\mathsf{x}}$ by adding a bar to the variable.

In Table 2.1 we overview all introduced theories. Next to the used abbreviation for the vocabulary and the structure of a theory, we provide the actual components of the corresponding vocabulary to ease lookup.

2.4 Higher order functions and relations

A recent trend supplements word equations, length- and regular membership constraints with higher order string operations. These operations are introduced within the SMT-LIB [111] standard and consists of commonly used operations used in programming languages to cope with strings. Within this section we introduce the used operations as abbreviations which make use of the operations discussed in the logical theory forming string constraints. Some of the newly operations contain existential quantifiers and therefore, due to negation within our formulae might introduce universal quantifiers. Consequently, even restricting certain theories to not cope with the full defined spectrum but the expressiveness of the following operations lead to new combined fragments not being as powerful as the whole combination described in the previous section but still lead to an undecidable sub theory. Many of these smaller combined theories are discussed in [41] and the references therein and classified in regards to their decidability status.

The first function at : $\mathtt{Pat}_A \times \mathbb{Z} \to A$ simply takes a pattern $\alpha \in \mathtt{Pat}_A$ and an integer $i \in \mathbb{Z}$ and returns a single constant of the unified pattern by an assignment $h \in \mathcal{H}_A$ at the i^{th} position, namely $h(\alpha)[i]$. If $i \notin [|h(\alpha)|]$ the function at simply returns the empty word ε. Formally we define this function by

$$\begin{aligned}
\mathrm{at}(\alpha, i) = \beta \text{ iff } (0 < i \leqslant \texttt{len}(\alpha) \rightarrow (\exists \mathrm{x}_1 \,.\, \exists \mathrm{x}_2 \,.\, \alpha \doteq \mathrm{x}_1 \cdot \beta \cdot \mathrm{x}_2 \\
&\wedge \texttt{len}(\mathrm{x}_1) = i - 1 \\
&\wedge \texttt{len}(\beta) = 1 \\
&\wedge \texttt{len}(\mathrm{x}_2) = \texttt{len}(\alpha) - i)) \\
((0 \leqslant i \vee i \geqslant \texttt{len}(\alpha)) \rightarrow \beta = \varepsilon),
\end{aligned}$$

for $\alpha \in \texttt{Pat}_A$, $i \in \mathbb{Z}$, and $\beta \in A$.

The second function substr : $\texttt{Pat}_A \times \mathbb{Z} \times \mathbb{Z} \rightarrow A^*$ takes a pattern $\alpha \in \texttt{Pat}_A$ and two integers $i, n \in \mathbb{Z}$ and returns a unified pattern $\beta \in A^*$ by a $h \in \mathcal{H}_A$ such that β is a factor of $h(\alpha)$ starting at the i^{th} position and having at most length n ($|\beta| \leqslant n$). Whenever the integer $i \notin [|h(\alpha)|]$ or $n \leqslant 0$ the function substr return ε ($\beta = \varepsilon$). Formally the function is defined by

$$\begin{aligned}
\mathrm{substr}(\alpha, i, n) = \beta \text{ iff } ((0 < i \leqslant \texttt{len}(\alpha) \wedge n > 0) \rightarrow (\exists \mathrm{x}_1 \,.\, \exists \mathrm{x}_2 \,.\, \alpha \doteq \mathrm{x}_1 \cdot \beta \cdot \mathrm{x}_2 \\
&\wedge \texttt{len}(\mathrm{x}_1) = i - 1 \\
&\wedge \texttt{len}(\mathrm{x}_2) = \max(\texttt{len}(\alpha) - ((i-1) + n)), 0) \\
\wedge\ ((0 \geqslant i \vee i \geqslant \texttt{len}(\alpha) \vee n \leqslant 0) \rightarrow \beta = \varepsilon),
\end{aligned}$$

for $\alpha \in \texttt{Pat}_A$, $i, n \in \mathbb{Z}$, and $\beta \in A^*$.

The next three relations are satisfied, whenever for two, by an assignment $h \in \mathcal{H}_A$ unified, patterns $\alpha, \beta \in \texttt{Pat}_A$, the pattern α is a factor, prefix, or suffix of β, respectively. Formally these relations contains : $\texttt{Pat}_A \times \texttt{Pat}_A$, prefixof : $\texttt{Pat}_A \times \texttt{Pat}_A$, and suffixof : $\texttt{Pat}_A \times \texttt{Pat}_A$ are defined as follows:

$$\begin{aligned}
\mathrm{contains}(\alpha, \beta) &\text{ iff } \exists \mathrm{x}_1 \,.\, \exists \mathrm{x}_2 \,.\, \beta \doteq \mathrm{x}_1 \cdot \alpha \cdot \mathrm{x}_2, \\
\mathrm{prefixof}(\alpha, \beta) &\text{ iff } \exists \mathrm{x}_1 \,.\, \beta \doteq \alpha \cdot \mathrm{x}_1, \text{ and} \\
\mathrm{suffixof}(\alpha, \beta) &\text{ iff } \exists \mathrm{x}_1 \,.\, \beta \doteq \mathrm{x}_1 \cdot \alpha,
\end{aligned}$$

for pattern $\alpha, \beta \in \texttt{Pat}_A$.

The function indexof : $\texttt{Pat}_A \times \texttt{Pat}_A \times \mathbb{Z} \rightarrow \mathbb{Z}$ takes two patterns $\alpha, \beta \in \texttt{Pat}_A$ and an integer $i \in \mathbb{Z}$ and returns the initial position of the first occurrence of $h(\beta)$ in $h(\alpha)$ after position i, if any, using a suitable substitution $h \in \mathcal{H}_A$. Whenever $h(\alpha)$ does not contain the factor $h(\beta)$ after position i the function indexof simply returns -1. Note, if $i \notin [|h(\alpha)|]$ we also return -1. Formally this function is defined by

$$\begin{aligned}
\text{indexof}(\alpha,\beta,i) = j \text{ iff } (\text{contains}(\beta,\alpha) \to \exists\,\mathsf{x}_1\,.\,\exists\,\mathsf{x}_2\,.\,\exists\,\mathsf{x}_3\,.\,\alpha = \mathsf{x}_1\cdot\mathsf{x}_2\cdot\mathsf{x}_3 \\
\wedge\ \mathtt{len}(\mathsf{x}_1) = i-1 \\
\wedge\ (\text{contains}(\beta,\mathsf{x}_2) \to (\exists\,\mathsf{x}_4\,.\,\exists\,\mathsf{x}_5\,.\,\mathsf{x}_2 \doteq \mathsf{x}_4\cdot\mathsf{x}_5 \\
\wedge\ \neg\text{contains}(\beta,\mathsf{x}_4) \\
\wedge\ \text{prefixof}(\beta,\mathsf{x}_5) \\
\wedge\ j = i + \mathtt{len}(\mathsf{x}_4))) \\
\wedge\ (\neg\text{contains}(\beta,\mathsf{x}_2) \to j = -1)) \\
\wedge\ (\neg\text{contains}(\beta,\alpha) \to j = -1),
\end{aligned}$$

for pattern $\alpha,\beta \in \mathtt{Pat}_A$ and $i,j \in \mathbb{Z}$.

The replace : $\mathtt{Pat}_A \times \mathtt{Pat}_A \times \mathtt{Pat}_A \to \mathtt{Pat}_A$ takes three pattern $\alpha,\beta,\gamma \in \mathtt{Pat}_A$. For an assignment $h \in \mathcal{H}_A$ we replace the first occurrence of $h(\beta)$ in $h(\alpha)$ by $h(\gamma)$, if $h(\beta)$ is a factor of $h(\alpha)$. Whenever $\beta = \varepsilon$, replace simply returns $\alpha\cdot\gamma$. Otherwise, if $h(\beta)$ is not a factor of $h(\alpha)$, we return α. Formally the replace function is defined by

$$\begin{aligned}
\text{replace}(\alpha,\beta,\gamma) = \delta \text{ iff } (\beta = \varepsilon \to \delta \doteq \alpha\cdot\gamma) \\
\wedge\ ((\neg\beta \doteq \varepsilon \wedge \neg\text{contains}(\beta,\alpha)) \to \delta \doteq \alpha) \\
\wedge\ ((\neg\beta \doteq \varepsilon \wedge \text{contains}(\beta,\alpha)) \to \\
(\exists\,\mathsf{x}_1\,.\,\exists\,\mathsf{x}_2\,.\,\alpha \doteq \mathsf{x}_1\cdot\beta\cdot\mathsf{x}_2 \\
\wedge\ \neg\text{contains}(\beta,\mathsf{x}_1) \\
\wedge\ \delta \doteq \mathsf{x}_1\cdot\gamma\cdot\mathsf{x}_2)),
\end{aligned}$$

for pattern $\alpha,\beta,\gamma,\delta \in \mathtt{Pat}_A$.

The next functions and relations describe conversions between pattern $\mathtt{Pat}_A$ and integers $\mathbb{Z}$. To ease readability of the following definitions instead of decimal numbers we restrict ourselves to binary numbers ranging over $\{\,0,1\,\}$. As commonly known this does not restrict expressiveness. Furthermore we assume the existence of a function code : $\textsc{Letters} \to \{\,0,1\,\}^*$ over a set of symbols $\textsc{Letters} \subseteq A$, mapping each $c \in \textsc{Letters}$ to the binary representation of a unique positive integer $i \in \mathbb{N}$ without leading zeros, i.e. $\textsc{Letters} = \{\,\mathsf{a},\mathsf{b},\mathsf{c}\,\}$ and code $= \{\,\mathsf{a} \mapsto 0, \mathsf{b} \mapsto 1, \mathsf{c} \mapsto 10\,\}$. Within the SMT-Lib Standard 2.6, which introduced the Unicode standard we address 196608 different characters. Therefore, in this case we have $|\textsc{Letters}| = 196608$.

The first relation isDigit : $\texttt{Pat}_A$ simply checks whether a pattern $\alpha \in \texttt{Pat}_A$ is a single constant which has a binary representation by using the function code. Formally we define the relation by

$$\text{isDigit}(\alpha) \texttt{ iff } \texttt{len}(\alpha) = 1 \wedge \alpha \mathrel{\dot\in} \mathsf{dom}(\mathsf{code}),$$

for a pattern $\alpha \in \texttt{Pat}_A$.

Since our goal is to use the previously discussed function code as a total function on all possible constants within our pattern $\alpha \in \texttt{Pat}_A$, as well as a conversions from a binary sequence to the corresponding character (whenever present in our alphabet $\textsc{Letters}$), we define two functions fromCode : $\mathbb{Z} \to A^*$ and toCode : $A^* \to \mathbb{Z}$. Thereby, fromCode maps a binary number $i \in \mathbb{Z}$ to its constant addressed by code and to the empty word ε, whenever the input i is not in the image of code. Consequently, toCode maps a pattern $\alpha \in \texttt{Pat}_A$ to its binary representation made by code and to -1 if $\alpha \notin \mathsf{dom}(\mathsf{code})$. Formally these functions are defined as follows:

$$\begin{aligned}\text{fromCode}(i) = \alpha \text{ iff } \quad & (i \geqslant 0 \wedge i \leqslant |\mathsf{dom}(\mathsf{code})| \to i = \mathsf{code}(\alpha)) \\ & \wedge (i < 0 \vee i > |\mathsf{dom}(\mathsf{code})| \to \alpha = \varepsilon)\end{aligned}$$

and

$$\begin{aligned}\text{toCode}(\alpha) = i \text{ iff } \quad & (\text{isDigit}(\alpha) \to i = \mathsf{code}(\alpha)) \\ & \wedge (\neg\text{isDigit}(\alpha) \to i = -1)\end{aligned}$$

for $\alpha \in \texttt{Pat}_A$, $i \in \mathbb{Z}$.

The remaining two functions, toInt : $\texttt{Pat}_A \to \mathbb{Z}$ and fromInt : $\mathbb{Z} \to \texttt{Pat}_A$ allow us converting a positive integer into its string representation, simply by using its binary representation over $\{0, 1\}$, and a string only consisting of digits within $\{0, 1\}$ (possibly having leading zeros) to its positive integer representation. Whenever the above mentioned criteria are not met, we return -1 and the empty word ε, respectively. To ease readability let $\mathsf{numstring} : \mathbb{N} \times \{0, 1\}^*$ be a relation, which contains all pairs of positive integers $i \in \mathbb{N}$ and words $w \in \{0, 1\}^*$ where w – again, possibly having leading zeros – is the binary representation of i. Formally this relation is defined by

$$\mathsf{numstring}(i, w) \text{ iff } w \mathrel{\dot{\in}} (0 \cup 1)^* \wedge |w| > 0 \wedge \sum_{j \in [|w|]} \mathsf{at}(w, j) \cdot 2^{|w|-j} = i.$$

Based on this we define

$$\begin{aligned} \text{toInt}(\alpha) = n \text{ iff } \quad & (\neg \alpha \doteq \varepsilon \wedge \alpha \mathrel{\dot{\in}} (0 \cup 1)^* \rightarrow \mathsf{numstring}(n, \alpha)) \\ & \wedge (\alpha \doteq \varepsilon \vee \neg \alpha \mathrel{\dot{\in}} (0 \cup 1)^* \rightarrow n = -1) \end{aligned}$$

and

$$\begin{aligned} \text{fromInt}(n) = \alpha \text{ iff } \quad & (n \geqslant 0 \rightarrow \mathsf{numstring}(n, \alpha)) \\ & \wedge (n < 0 \rightarrow \alpha = \varepsilon) \end{aligned}$$

for a pattern $\alpha \in \mathtt{Pat}_A$ and $n \in \mathbb{Z}$.

Remark 2.16. As mentioned before, in practice the conversions functions do not convert to binary but decimal representation. To ease readability we defined the above functions to cope with binary numbers.

Chapter 3

String Constraints in Practice

> *"The fact that we survive at all is kind of a surprise."*
>
> RVIVR

Stated in [8] and the references therein the first applications of string constraint solving goes back to the late 1980s, the time where *constraint programming* [98] arose, by the authors of TRILOGY [115] – a language providing strings, integer, and real constraints. After many years of stagnation and only minor interest into the topic, in 2010 string constraints started playing a central role within the formal verification community. In this chapter we will cover the most general applications of string constraints, solving strategies, and commonly used solvers.

3.1 Verification of String Inputs

Nearly any broadly used programming language deals with the datatype of *strings* and operations such as comparison, membership and manipulation of strings. Consequently analysing nowadays software involves the investigation of this datatype which strictly leads us to the need of ensuring correct handling of strings. Following [29] the common uses of strings in program verification are input sanitisation and validation (cf. [90]), query generation for databases (cf. [56]), and dynamic code and data generation (i.e XML, JSON, and HTML) (cf. [92]). These use cases are closely related to each other: most modern software applications allow users to provide input via forms. Based on the input the application generates database queries, data being presented within the application,

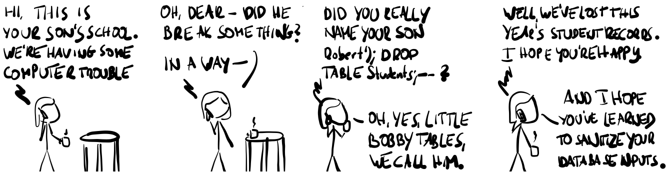

HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE
OH, DEAR - DID HE BREAK SOMETHING?
IN A WAY -)
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH, YES, LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Figure 3.1. XKCD [119] comic visualising an SQL injection

or in a server-client side scenario of web applications even code which is executed on the users devices. As user input always gives the opportunity of gaining unwanted access, submitting malicious or removing data from the servers, avoiding errors within applications using strings is crucial. Unfortunately, like many other program analysis problems, the problem of determining the set of all unwanted inputs is undecidable [29]. In general identifying potential security vulnerabilities within these tasks is called *string analysis* within the formal verification community.

Taking a closer look at all reported security related vulnerabilities listed in the Common Vulnerabilities and Exposures Repositories (CVE) [37] two of the most occurring issues are related to strings. That are *Cross-site Scripting* and *SQL injection*. For a Cross-site Scripting attack an attacker inserts malicious data into an HTML website. A rather common attack in this regards is publishing a link pointing to a malicious JavaScript file which is executed on the victims client whenever clicked and possibly revealing sensitive data to the attacker. A more concrete example is given in Listing 3.1 originally taken from [29].

```
<?php
 $www = $_GET["www"];
 $l_otherinfo = "URL";
 $www = preg_replace("/[^A-Za-z0-9␣.-@:/]/", "", $www);
 echo $l_otherinfo . ":␣" . $www;
?>
```

Listing 3.1. A server-side input sanitisation code snippet in PHP taken from [29]

The example shows vulnerable code from a web application called MyEasyMarket [15] and was used to sanitize user's input. In line 2 the application fetches an external parameter called `www` into the variable `$www`. Within the next line we assign a constant string `URL` to a variable `$l_otherinfo`. Line 5 is supposed to sanitize the input stored in `$www`. The PHP function `preg_replace` replaces every string matching a regular language with a given string. Within the example every string within the variable `$www` matching the regular expression `/[^A-Za-z0-9 .-@:/]/` will be erased (replacement by the empty string). In line 5 we output URL and the sanitized input of the variable `$www`. The aforementioned error lies within the sanitisation step made in line 5: the stated regular expression does not meet the developers intention of removing everything other than alphanumeric characters and the symbols `.`, `-`, `@`, `:`, and `/`. The developer missed the special semantics of `-` within a regular expression. Instead of listing all unwanted symbols by `.-@`, we specified the union of all symbols lying between `.` and `@` within the present order. Since `<` is within this range an attacker has the possibility of invoking HTML-code to gather private data. The correct regular expression using proper escaping has the following form `/[^A-Za-z0-9 .\-@:/]/`.

An SQL injection attack describes the automatic generation of a database query based on a user's input which contains possibly malicious data. A well known example depicted in Figure 3.1 was published by XKCD [119]. The user's input

```
Robert'); DROP TABLE Students;--
```

finishes an internal `SELECT` query and executes the removal of all students data. The previous attack scenarios, also the on visualised in Figure 3.1, are caused by insufficient validation and sanitisation of the input and again show the need of a proper input analysis. Especially the error within the regular expression in Listing 3.1 is extremely hard to spot. This calls for proper automatic string analysis mechanisms.

Due to the ever growing importance of strings many analysis techniques have been developed – still often not used as we discussed previously.

In the following we highlight the common used techniques:

1. automata-based (cf. [33]) string analysis uses the source code and builds an over approximation of the program using a symbolic finite automaton per string variable to detect possible vulnerabilities. In that sense each finite automaton represents the set of possible solutions to a string variable.
2. relational string analysis extends the automata-based approach by the ability of tracking relationships between variables by using a multi-track-automata, each track modelling a single string variable instead of a simple finite state automaton per variable not being coupled, and
3. string constraint solving [8] which verifies the satisfiability of a formula in first order logic being a representation of the source code. The formula is obtained by performing symbolic execution on a string manipulating program.

Since string constraint solving builds the core of the presented work we will focus on this type of string analysis. The next section introduces the commonly used underlying mechanisms for solving string constraints.

3.2 Underlying Mechanisms for Solving String Constraints

According to [8] solving formulae involving string constraints is primarily achieved by using *constraint logic programming* (CP) [98] and *satisfiability modulo theories* (SMT) [20] solvers. Within this section we briefly introduce both mechanisms and their extensions to deal with string constraints.

3.2.1 Strings in Constraint Logic Programming

Constraint Logic Programming was first introduced in the 1970s within the context of artificial intelligence [53] but found its first practical applications on scalable systems in the late 1990s accommodating tools like Gecode [101] and Chuffed [35]. Within CP solving we ask whether a *constraint satisfaction problem* (CSP), a structure consisting of variables, finite variable domains and constraints over the given variables, is satisfiable. A CSP is satisfiable if we are able to find an assignment for all

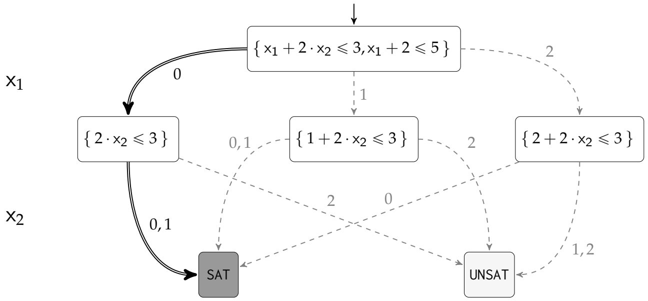

$\{ x_1 + 2 \cdot x_2 \leqslant 3, x_1 + 2 \leqslant 5 \}$
x_1
0
1
2
$\{ 2 \cdot x_2 \leqslant 3 \}$
$\{ 1 + 2 \cdot x_2 \leqslant 3 \}$
$\{ 2 + 2 \cdot x_2 \leqslant 3 \}$
0, 1
2
x_2
0, 1
2
0
1, 2
SAT
UNSAT

Figure 3.2. Solving the linear inequalties given in Example 3.1

variables mapping to values within the attached domains such that the constraints are satisfied. More formally speaking a CSP is a triple (X, D, C) consisting of a tuple of variables $X = (x_1, \dots, x_n)$ such that $x_i \in \mathcal{X}$ for $i \in [n]_0$, a finite tuple of domains $D = (D_1, \dots, D_n)$ for the variables X, and a finite set of constraints $C = \{ C_1, \dots, C_m \}$ for $n, m \in \mathbb{N}$. Moreover, each constraint $C_i = (S_i, R_i)$ is a tuple consisting of $S_i = (y_{i_1}, \dots, y_{i_\ell})$ with $0 < \ell \leqslant n$ and there exists $k \in [n]$ such that $y_{i_j} = x_k$ for all $j \in [\ell]$ the tuple of variables appearing in C_i and R_i an $|S_i|$-ary relation over the variables of S_i. A mapping $h : X \to \bigcup D$ such that $h(x_i) \in D_i$ for all $i \in [n]$ is called a *solution* to (X, D, C) if for all $(S_i, R_i) \in C$ we have $(h(y_1), \dots, h(y_\ell)) \in R_i$. The set of constraints is usually stated implicitly, e.g. for variables x_1, x_2 both having the domain [3] we simply write $x_1 = x_2$ instead of $(\{ x_1, x_2 \}, \{ (0,0), (1,1), (2,2), (3,3) \}$. The general mechanism of a CP solver is the combination a basic backtracking while keeping track of a partial solution and the propagation of gained knowledge to prune the domains by removing infeasible values.

Example 3.1. Given a CSP $P = ((x_1, x_2), ([2], [2]), \{ x_1 + 2 \cdot x_2 \leqslant 3, x_1 + 2 \leqslant 5 \})$. In Figure 3.2 we depict the graph dealing within all possible assignments with our given domains by first branching on variable x_1 and afterwards on x_2. A CP solver initially selects an assignment for a variable, let us say it sets $h(x_1) = 0$. Afterwards he propagates the reduced linear inequality which immediately leads to a reduction of possible assignments for x_2 within the given domain, namely $\{ 0,1 \}$. The solver afterwards simply selects one of the values and, i.e. returns $h = \{ \mathsf{x}_1 \mapsto 0, \mathsf{x}_2 \mapsto 1 \}$ which is indeed a solution to the CSP.

We have $h(\mathsf{x}_1) = 0 \in [2]$, $h(\mathsf{x}_1) = 1 \in [2]$,

$$h(\mathsf{x}_1) + 2 \cdot h(\mathsf{x}_2) = 0 + 2 \cdot 1 = 2 \leqslant 3,$$

and

$$h(\mathsf{x}_1) + 2 = 0 + 2 \leqslant 5.$$

Therefore $h \models P$.

The above example looks similar to Example 2.8 seen within the introduction of our logical theories. In fact all that is required additionally to encode linear (in)equalities into a CSP are the finite domains.

Let (X, D, C) be a CSP. To cope with strings (cf. [9] and the references therein) we define an equality predicate over our terminal alphabet A^* as seen in Section 2.2.1. Considering all possible solution to the variables x_i in X leads to an infinite domain for all variables within the tuple X, i.e. the domain A^*. Therefore, a classical approach is either defining an exact bound k (i.e. A^k) or an upper bound k (i.e. $A^{\leqslant k} = \{ w \in A^* \mid |w| \leqslant k \}$) for some $k \in \mathbb{N}$. This naturally leads to finite domains. The actual solving of a CSP is now done in exactly the same way as described previously - we prune the domain of possible solutions by propagating gathered knowledge.

Example 3.2. Let $A = \{ a, b, c \}$ and

$$P = ((\mathsf{x}_1, \mathsf{x}_2), (A^{\leqslant 2}, A^{\leqslant 2}), \{ a\mathsf{x}_1 ab\mathsf{x}_2 ab\mathsf{x}_1 b \doteq \mathsf{x}_2 ba\mathsf{x}_1 aabb\mathsf{x}_1 \})$$

a CSP. Since we are not able to propagate knowledge at this initial position in the search, we assume the solvers starts branching on x_1. Following the assignment $h(\mathsf{x}_1) = \varepsilon$ reveals the word equation $aab\mathsf{x}_2 abb \doteq \mathsf{x}_2 baaabb$. A smart propagator is know able to prune the domain of x_2 to the set $\{ aa \}$, since any other assignment within $A^{\leqslant 2}$ directly leads to an unsatisfiable substitution of x_2. Thus, by following this path, we obtain the solution $h(\mathsf{x}_1) = \varepsilon$ and $h(\mathsf{x}_2) = aa$.

The main drawback of CP approaches with respect to string solving is the finite domain which needs to be fixed in advanced. On the good end this naturally leads to decidability even if considering the combination of all fragments presented in the previous chapter. Moreover, the finite domains directly lead to an NP upper bound.

3.2.2 Strings in Satisfiability Modulo Theories

Within the propositional logic satisfiability problem (see Section 2.1.1) we aim to decide whether a formula containing propositional logic variables and classical connectives is satisfiable or not. In our case (String Constraint Solving) and also many other problems we relay on a richer language to express our goals. This is where satisfiability modulo theories (SMT) comes into play. The key insight is to incorporate theories (as seen in Section 2.2) into the classical DPLL algorithm [39] by Davis, Putnam, Logemann, and Loveland to support a more expressive input language. To this extend most modern SMT solvers implement an extension of this well known algorithm called DPLL(T) [93]. The core idea is the interaction of the theory solvers which decompose their parts of the input formula into a propositional logic satisfiablity problem by building an over approximation, asking the core SAT solver about the satisfiability and a model, respectively, of their abstraction, and then checking feasibility of the original formula [44]. Therefore, word equations get their attention within a sub solver handling exactly the theory introduced in Section 2.2.1.

Example 3.3. We will review the interleaving within a DPLL(T) SMT-solver and the general behaviour. Consider a variant of the Example 2.14 consisting of the word equation $a\mathsf{x}_1 ab\mathsf{x}_2 ab\mathsf{x}_1 b \doteq \mathsf{x}_2 ba\mathsf{x}_1 aabb\mathsf{x}_1$ and the length constraint $\mathsf{len}(\mathsf{x}_1) \cdot 2 \leqslant \mathsf{len}(\mathsf{x}_2)$. Firstly, the solver builds a propositional logic abstraction to check whether the formula is – in principle – satisfiable. In that sense each atom of the formula is represented by a propositional logic variable. In our setting this leads to the introduction of two propositional logic variables b_1 and b_2 representing the word equation and the length constraint, respectively. The conjunction of both constraints lead to the resulting formula $\mathsf{b}_1 \wedge \mathsf{b}_2$ which is satisfiable if both constraints are satisfied. Secondly, the constraints which need to be satisfied according to the model produced for the abstraction are distributed to their theory solvers. Since in our example both constraints has to be satisfied we distribute the word equation to the string solving theory solver and the linear inequalty to the arithmetic solver. The theory solvers try to deduce an assignment.

During the search, the theory solvers assert so called theory lemmas back to the core algorithm which triggers the initially discussed abstraction and distribution of atoms again. Our string solver can for example deduce a theory lemma stating $\text{len}(x_2) \geqslant 2$ which is then distributed to the arithmetic solver. This interleaving is repeated iteratively until we either deduce the satisfiability, unsatisfiability, or run into a resource limit.

As also seen in Section 2.2 considering combinations of the discussed theories might lead to an undecidable satisfiability problem, (i.e. the combination of word equations, (in)equalties, and regular membership constraints). Given this, termination is not guaranteed when dealing with these theories, which also forms one of the major disadvantages when comparing CP and SMT approaches.

Within the next section we introduce several state of the art string solvers and embed them with respect to their solving strategy.

3.3 String Constraint Solvers

Since string constraints arose practical attention there had be numerous attempts of solving them in a efficient way. In general all approaches fall into one of three categories: 1. automata-based, 2. word-based, and 3. unfolding-based [8]. Within the first approach each variable is represented by a finite automaton which describes suitable solutions. The formula in question maps directly to the respective automaton. The second approach reasons mostly algebraically on a particular constraint, centralising the structure of words while determining satisfiability. Knowledge obtained from theoretical observations of particular theories are directly applied to the input formula. Unfolding-based approaches are characterised by encoding string constraints into well established formats where efficient solvers exist. These are for example propositional logic formulae, arithmetic expression, or bit-vectors.

Each of the introduced category can be divided into three strategies targeting the length of a solution of a particular variable – A solver either assumes 1. a fixed length, 2. an upper bound, or 3. an unbounded length for the solution of the variables. As discussed at the end of Section 3.2.1 having an upper bound on the length of the solution simplifies the solving

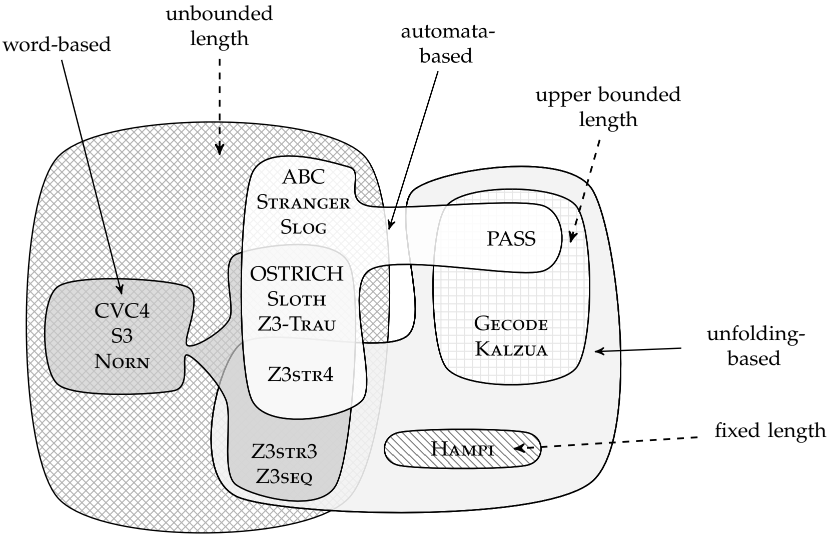


Figure 3.3. Grouping of established solvers based on their behaviour

process drastically but a solver might miss potential solutions larger than the given bounds.

In practice, all variants are not only used solely but also a combination of the presented approaches. The most prominent tools (according to the literature) implementing a word-based approach are CVC4 [16], S3 [113], and Norn [4] which are all SMT-solvers reasoning about unbounded length solutions for the variables. The algorithm of CVC4 is based on derivation rules which are iteratively applied until unsatisfiability is detected or no further rule can be applied. In the latter case, also called *saturated configuration*, the input formula is satisfiable. S3 is build on top of Z3-str [123] which was the first module to solving string constraints in Z3 [45], a general purpose SMT-solver. The strategy of Z3-str is treating strings as primitive types. Each string of constants is systematically broken into substrings, while variables are split into multiple variables until a solved form is reached. S3 extends Z3-str to cope with regular membership predicates and higher level functions used in program analysis. NORN performs a depth-first search on the given input formula. During the search the solver applies a sequence of rules, first checking the arithmetic constraints, followed by an elimination of inequalities, a splitting procedure for word equations as well as for regular membership constraints to ease the search. In the end it performs a satisfiability check of the remaining constraints.

In the category of unfolding based approaches we find GECODE [101], KALUZA [99], and HAMPI [71]. Except for HAMPI, which assumes fixed length variables, the other tools require an upper bound on the length of the solution. Both, HAMPI and KALUZA encode string constraints directly into bit-vector. GECODE, the only CP-solver for string constraints we overview, uses the concept of *dashed strings*, which are concatenations of sets of strings to minimise the domain size for the solver.

Within the automata-based approach the solvers mentioned most often are STRANGER [120], ABC [11], and SLOG [116]. All tools also reason on unbounded length. The tool STRANGER is not a string solver but integrates a technique for solving string constraints within the static analysis of PHP applications. It implements an automata-based symbolic analysis of string constraints by computing all possible values a string expression might have. The tool SLOG is build around NFA manipulations represented by logic circuits. To this extend they avoid determinisation as long as possible and purely relay on the underlying logic circuits for solving string constraints. ABC implements a model counting approach, by iteratively refining an automaton it describes all potential solutions to the considered input formula.

Next to these solvers, there exist many other approaches which use interleaving techniques from the above mentioned categories. The tool PASS [79] only uses automata to speed up the search for a solution of word equations and regular expression membership constraints. The key architecture relays on an encoding into parametrized arrays. The solver does not reason upon unbounded length but requires an upper bound.

The solver Z3-TRAU [1], SLOTH [62], and OSTRICH [30] are build upon SMT-solvers but use automata to represent certain structures. Z3-TRAU uses parametric flat automata to deal with string constraints. SLOTH [62] and OSTRICH [30], which is a successor of SLOTH [62], use alternating finite state automata to handle string constraints.

On the other end the SMT-solvers Z3str3 and Z3seq combine word-based and unfolding-based approaches. Z3seq reduces word equations to the theory of sequences and characters. Secondly, it uses Brzozowski derivatives to solving regular expression membership constraints. Z3str3 uses a technique called *theory aware branching*. It applies a technique similar to the rules induced by Levi's lemma (called arrangements) to simplify an equation. The resulting word equations are rated according to their complexity. Z3str3 prioritises simpler word equations over harder ones in order to speed up the search. Secondly, it uses a fixed length reduction to bit-vectors to construct the resulting model for a given string constraint.

An exception in all of these categories is a solver called Z3str4 [25]. This solver can be seen as portfolio solver, since it leverages different solving strategies implemented within Z3, namely Z3str3 [23], Z3seq [45], and a novel implementation of the length abstraction solver which is an implementation of filling the positions approach (details will be discussed in Section 3.5.1) combined with a blocking of used bounds based on previously discovered counter-examples. Z3str4 uses four different sequences of solvers (called arms). The arms are selected based on structural properties of the input formula via probes. Notably, the solver uses the technique for solving regular membership constraints we establish in Section 4.3.

We overview all mentioned solvers in Figure 3.3.

Nowadays, the most used solvers in literature and industrial applications are Z3str3 and Z3seq both being part of the Z3-family [45] developed by Microsoft Research and CVC4 and open source project lead by Clark Barrett and Cesare Tinelli. Their active development and efficient algorithms make them unavoidable in the area of string solvers. Since both solvers build up on an SMT infrastructure the developers had a personal interest in developing a uniform input format. The authors of CVC4 made a first attempt in establishing a uniform language in 2010 [18, 19] which was not accommodating strings. After a meeting of developers of Z3 and CVC4 at MOSCA 2019 [58] the SMT-LIB standard in version 2.6 was released in early 2020 [111] incorporating formal definitions for string constraints. The SMT-LIB standard incorporates the theory seen in Section 2.3.3 together with the associated higher order functions introduced in Section 2.4. In Figure 3.4 we provide a grammar for the basic operations

$$
\begin{array}{lll}
F & ::= & Atom \mid (\texttt{and}\ F\ F) \mid (\texttt{or}\ F\ F) \mid (\texttt{not}\ F) \\
Atom & ::= & (\texttt{=}\ t_{str}\ t_{str}) \mid A_{int} \mid A_{ext} \mid (\texttt{str.in\_re}\ t_{str}\ RE) \\
A_{int} & ::= & (\texttt{=}\ t_{int}\ t_{int}) \mid (\texttt{<}\ t_{int}\ t_{int}) \\
A_{ext} & ::= & (\texttt{str.contains}\ t_{str}\ t_{str}) \mid (\texttt{str.prefixof}\ t_{str}\ t_{str}) \mid \\
 & & (\texttt{str.suffixof}\ t_{str}\ t_{str}) \\
t_{int} & ::= & m \mid \bar{\mathtt{x}} \mid (\texttt{str.len}\ t_{str}) \mid (\texttt{+}\ t_{int}\ t_{int}) \mid (\texttt{*}\ m\ t_{int}) \mid \\
 & & (\texttt{str.indexof}\ t_{str}\ t_{str}\ t_{int}) \mid \\
 & & (\texttt{str.to\_int}\ t_{str}), \\
 & & \text{with } m \in \mathbb{Z} \text{ and } \bar{\mathtt{x}} \in \mathcal{X} \\
t_{str} & ::= & \text{“}w\text{”} \mid \mathtt{x} \mid (\texttt{str.++}\ t_{str}\ t_{str}) \mid (\texttt{str.from\_int}\ t_{int}) \mid \\
 & & (\texttt{str.replace}\ t_{str}\ t_{str}\ t_{str}) \mid (\texttt{str.at}\ t_{str}\ t_{int}) \mid \\
 & & (\texttt{str.substr}\ t_{str}\ t_{int}\ t_{int}), \\
 & & \text{with } w \in A^* \text{ and } \mathtt{x} \in \mathcal{X} \\
RE & ::= & \text{“}w\text{”} \mid (\texttt{re.none}) \mid (\texttt{re.++}\ RE\ RE) \mid (\texttt{re.comp}\ RE) \mid \\
 & & (\texttt{re.union}\ RE\ RE) \mid (\texttt{re.inter}\ RE\ RE) \mid (\texttt{re.*}\ RE), \\
 & & \text{with } w \in A^*
\end{array}
$$

Figure 3.4. Basic SMT-LIB syntax for string constraints.

within the string standard of SMT-LIB 2.6. All operations directly map to the ones introduced in Section 2.4, which also defines their semantics.

Within the next section we will observe how source code involving the data type of strings can be translated into a first order logic formula. The resulting formula will mostly be formatted in SMT-LIB to directly involve one (or more) of the aforementioned string solvers to search for vulnerabilities.

3.4 Source Code Verification Involving Strings

We briefly mentioned in Section 3.1 that in order to obtain a formula based on a programs source code we preform a program analysis called symbolic execution [72]. We will coarsely cover the intuition of this technique and will not elaborate the formal background but give a sophisticated introduction by using a concrete example. Symbolic execution is a technique allowing us to determine what sets of values cause a program to execute a particular path. Instead of trying different concrete values it assumes symbolic ones as inputs and expressions encountered during a symbolic run will be treated as functions of the corresponding symbolic variables. A particular location of a symbolic run is represented by the program counter and a path condition accommodating the symbolic variables. The path condition corresponding to a location is a constraint which must be satisfiable whenever we are able to reach the programs location. A graph representing all possible executions of a program using path conditions is called symbolic execution tree.

In the following we consider a program written in Python 3 to get an impression on how a symbolic execution leads us to a formula consisting of string constraints.

```
1  def exec_prog(cmd):
2    p = "/servers/dir/to/bin"
3    i = -1 if not "/" in cmd else cmd.rindex("/")
         ↪ # referenced as cmd.indexof("/")
4    if i == -1:
5      r = cmd
6    else:
7      r = cmd[i:]
8    t = r.startswith("rm")
9    if t:
10     raise RuntimeError("You’re not allowed to
           ↪ remove things!")
11   exec(p + r)
```

Listing 3.2. Example Python 3 program to remotely execute a program

In Listing 3.2 we present a function influenced by an example given in [29] which executes a binary on the server running the above script based on a user's input. After initialising a constant `p` to the string `/servers ↪ /dir/to/bin` in line 2 we initialise a variable `i` to the index of the last occurrence of a slash. Doing this gives us the index within the string to the passed path to the binary. If the passed string does not contain a slash we simply set `i` to -1. Afterwards, starting in line 4 we either set the variable `r` to the suffix of the passed string, starting at position `i` or just use the passed input. We want to disallow to execute the program `rm`. Therefore in line 8 we check whether our input binary has a prefix `rm`. An attempt to

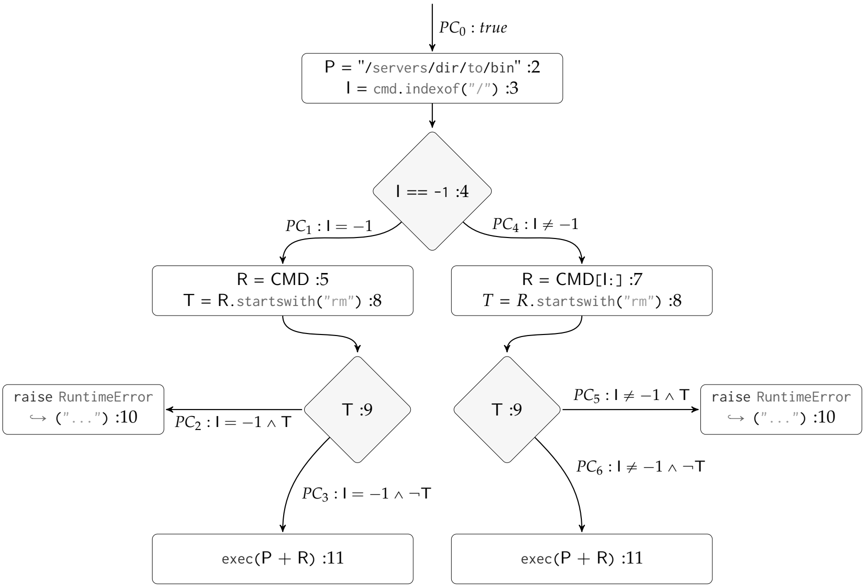

$PC_0 : true$
P = "/servers/dir/to/bin" :2
I = cmd.indexof("/") :3
I == -1 :4
$PC_1 : \mathsf{I} = -1$
$PC_4 : \mathsf{I} \neq -1$
R = CMD :5
T = R.startswith("rm") :8
R = CMD[I:] :7
T = R.startswith("rm") :8
T :9
T :9
raise RuntimeError ↪ ("...") :10
$PC_2 : \mathsf{I} = -1 \wedge \mathsf{T}$
$PC_5 : \mathsf{I} \neq -1 \wedge \mathsf{T}$
raise RuntimeError ↪ ("...") :10
$PC_3 : \mathsf{I} = -1 \wedge \neg\mathsf{T}$
$PC_6 : \mathsf{I} \neq -1 \wedge \neg\mathsf{T}$
exec(P + R) :11
exec(P + R) :11

Figure 3.5. Symbolic execution tree corresponding to Listing 3.2

execute `rm` results in an exception given in line 10. Otherwise the binary stored in `r` is executed on the server.

Performing a symbolic execution of the program reveals the symbolic execution tree depicted in Figure 3.5. Within this figure we represent the program variables with symbolic variable indicated by uppercase letters (e.g. `cmd` by CMD). Rectangle nodes represent expressions updating the variables and diamond nodes correspond to branching points within the program. The matching line numbers are written after each condition, or, respectively expression. By PC_i for $i \in [6]$ we reference the path condition which is updated while traversing the graph. The graph reveals four feasible runs guarded by the path conditions PC_2, PC_3, PC_5, and P_6 where $PC_2 : \mathsf{I} = -1 \wedge \mathsf{T}$ and $PC_5 : \mathsf{I} \neq -1 \wedge \mathsf{T}$ guard the exception and therefore the possible exploits which we want to avoid. Analysing these path constraints by unfolding their corresponding functions and predicates leads to our first order logic formula

$$\text{indexof}(\mathsf{CMD}, /, 1) = -1 \land \mathsf{R} \doteq \mathsf{CMD} \land \text{prefixof}(rm, \mathsf{R}) \tag{PC_2}$$

and

$$\begin{aligned}&\text{indexof}(\mathsf{CMD}, /, 1) \neq -1\\ &\land \mathsf{R} \doteq \text{substr}(\mathsf{CMD}, \text{indexof}(\mathsf{CMD}, /, 1), \mathtt{len}(\mathsf{CMD}))\\ &\land \text{prefixof}(rm, \mathsf{R}).\end{aligned} \tag{PC_5}$$

Note, we directly used the higher order functions introduced in Section 2.4 and as we observed there many of these functions are extremely difficult to solve. Thus, even for a toy example we clearly see the need for a proper string analysis.

As we will show later within this work many benchmarks are directly constructed by using a symbolic execution tool to obtain test data. A rather prominent example is the tool PyEx [13] – a symbolic executor for Python programs – used by Reynolds et al. [97] to generate a hard to solve set for state of the art string solvers (named after the tool – PyEx) of real-life benchmarks.

3.5 Solving Strategies

In the nature of our many-sorted structure of string constraints we naturally develop algorithms which target a specific theory, i.e word equations, linear length constraints or regular membership constraints. In the setting for SMT-solvers there is a permanent interleaving and knowledge sharing between the different solving strategies, e.g. an algorithm for solving word equations only needs to consider a certain subset of solutions if there a restrictions by regular membership constraints on certain string variables. Within this section, we cover algorithms which directly influenced the presented work, that are in particular two techniques for solving word equations. Both algorithms for solving word equations stem from lemmas stated within the theory of combinatorics on words.

3.5.1 Filling the positions

Plandowski [94] showed an upper bound of $2^{2^{O(n^4)}}$ on the length of the shortest solutions of a word equation of length n. It is, however, a widely accepted conjecture that the length of the shortest solution of a word equation is, in fact, exponential in n (which would imply, according to [95], that the satisfiability of word equations is in NP). In this context, one of the standard methods for solving word equations is the technique of *filling the positions* (see [68, 95]). Essentially, this consists of non-deterministically fixing the length of variables, and then arranging the individual positions of a solution, as referenced by their origin (e.g. the third letter of the variable x) into equivalence classes connecting them, either by their occurrence on the same position in different copies of the same variable, or by their occurrence on corresponding positions on the two sides of the equation.

Let $e = \alpha \doteq \beta \in \mathsf{WEQ}_A$ and $\mathtt{vars}\,(e) = \{\, \mathsf{x} \mid |\alpha\beta|_{\mathsf{x}} \neq 0 \,\}$ be the set of string variables occurring within the equation e. The algorithm assumes the existence of a mapping $z : \mathtt{vars}\,(e) \to \mathbb{N}$ assigning a length to each variable occurring within the equation e. Furthermore, let $\mathsf{fLen}_z(\alpha) : \mathtt{Pat}_A \to \mathbb{N}$ defined by

$$\mathsf{fLen}_z(\alpha) = \begin{cases} z(\mathsf{x}) & \text{if } \alpha = \mathsf{x} \in \mathcal{X}, \\ 1 & \text{if } \alpha \in A, \\ \sum_{i \in [\ell]_0} \mathsf{fLen}_z(\alpha[i]) & \text{if } |\alpha| = \ell \in \mathbb{N}_{>1}, \end{cases}$$

be a function calculating the length of a pattern being *filled* with respect to z. Based on this function we define another function having the ability of querying a specific position of a pattern under z, namely $\mathsf{fPos}_z^{\alpha} : [\mathsf{fLen}_z(\alpha)]_0 \to \mathbb{N} \times (A \cup \mathcal{X})$ defined by the following equivalence

$$\mathsf{fPos}_z^{\alpha}(i) = (j, x) \text{ iff } \exists\, p \in [\mathsf{fLen}_z(\alpha)]\,.\, \mathsf{fLen}_z(\alpha[1:p]) < i \leqslant \mathsf{fLen}_z(\alpha[1:p+1]) \\ \wedge\, j = i - \mathsf{fLen}_z(\alpha[1:p]) \wedge \alpha[p+1] = x.$$

We shall call the sequence obtained by $\mathsf{fPos}_z^{\alpha}(1) \ldots \mathsf{fPos}_z^{\alpha}(\mathsf{fLen}_z(\alpha))$ the *filled pattern under z*. The next step is defining an equivalence relation

$R_z \subseteq \mathbb{N} \times \mathbb{N}$ as follows

$$\begin{aligned}(i,j) \in R_z \text{ iff } &\mathsf{fPos}_z^{\alpha}(i) = \mathsf{fPos}_z^{\beta}(j)\\ &\vee \mathsf{fPos}_z^{\alpha}(i) = \mathsf{fPos}_z^{\alpha}(j)\\ &\vee \mathsf{fPos}_z^{\beta}(i) = \mathsf{fPos}_z^{\beta}(j).\end{aligned}$$

Two positions of a filled pattern are in relation R_z whenever they reference the same variable at the same position or the same constant letter. Using the relation R_z we are able to obtain an assignment h such that $|h(\mathsf{x})| = z(\mathsf{x})$ for each $\mathsf{x} \in \mathtt{vars}\,(e)$ by considering the equivalence classes of R_z. To acquire the corresponding constants to a position let $\mathsf{constants}(i) = \{\, a \mid a = \pi_2(\mathsf{fPos}_z^{\alpha}(j)) \in A, j \in [i]_{R_z} \,\}$ denote the constants corresponding to a single equivalence class. The word equation e only has a solution with fixed length z if $|\mathsf{constants}(i)| \leqslant 1$ holds for all $i \in [\mathsf{fLen}(\alpha)]_0$. Moreover, if the condition holds, we are able to acquire an assignment for each variable $\mathsf{x} \in \mathtt{vars}\,(e)$ such that $h(\mathsf{x}) = \gamma \in A^*$ and $|\gamma| = z(\mathsf{x})$ by

$$\gamma[i] = \begin{cases} a & \text{if } j \in [\mathsf{fLen}(\alpha)]_0, \mathsf{fPos}_z^{\alpha}(j) = (i,\mathsf{x}), \{\, a \,\} = \mathsf{constants}(j),\\ b & \text{else for any arbitrary } b \in A.\end{cases}$$

A proof of the correctness of the induced algorithm is given in [68]. This definition might seem complex but the intuition behind it is surprisingly simple as we highlight in the follow example.

Example 3.4. Consider the word equation

$$\alpha = a\mathsf{x}_1ab\mathsf{x}_2ab\mathsf{x}_1b \doteq \mathsf{x}_2ba\mathsf{x}_1aabb\mathsf{x}_1 = \beta$$

and the mapping $z = \{\, \mathsf{x}_1 \mapsto 1, \mathsf{x}_2 \mapsto 3 \,\}$. Therefore we have

$$\mathsf{fLen}_z(\alpha) = 11 = \mathsf{fLen}_z(\beta).$$

By calculating fPos_z^{α} and fPos_z^{β} for all $i \in [\mathsf{fLen}_z(\alpha)]_0$, respectively, we obtain the structure given in Figure 3.6. With respect to R_z this leads to two equivalence classes, namely $[1]_{R_z} = \{\, 1,3,5,7,8 \,\}$ and $[2]_{R_z} = \{\, 2,4,6,9,10,11 \,\}$ corresponding to a and b, respectively. We now obtain the solution $h = \{\, \mathsf{x}_1 \mapsto b, \mathsf{x}_2 \mapsto aba \,\}$ visualised in Figure 3.7. We are able to observe that both sides of the word equation have been unified correctly.

Filling the position gives us a straight forward algorithm to solving

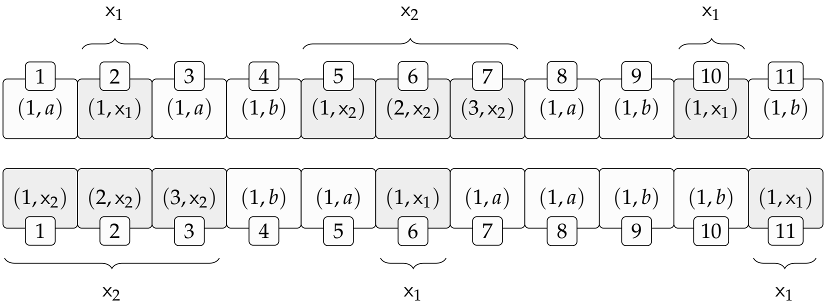


Figure 3.6. Initial setup for the filling the poistions algorithm

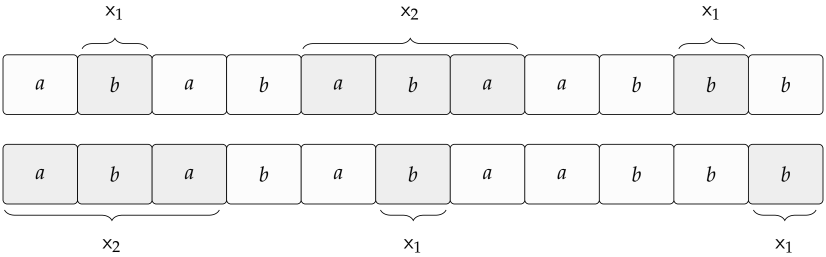

x_1
x_2
x_1
a b a b a b a a b b b
a b a b a b a a b b b
x_2
x_1
x_1

Figure 3.7. Propagated constants within the given word equation

word equations whenever we know about the exact bounds of the variables being present. In practice, bounds are mostly not known but as we have seen within the theory of word equations containing linear length constraints, variables are often upper bounded which makes a stepwise approximation to the given upper bounds possible and the presented algorithm feasible in practice.

3.5.2 Levi's Lemma

A transformation-system for solving systems of word equations is induced by *Levi's lemma* [78], a classical result in the combinatorial theory of words (sometimes also called the Nielsen transformation, by analogy to a technique from combinatorial group theory). The lemma as used nowadays is given as follows:

Lemma 3.5 (Levi's lemma [32]). *Let $u, v, x, y \in \mathtt{Pat}_A$. If $uv = xy$ then there exists a word $t \in \mathtt{Pat}_A$ such that $u = xt$ and $tv = y$, or $x = ut$ and $v = ty$.*

Informally Levi's lemma non-deterministically guesses on the prefixes of the two sides of the word equation, and therefore introduces a rewriting relation $\rightarrow \ \subseteq \mathtt{Pat}_A \times \mathtt{Pat}_A$. The goal is to introduce a set of rewriting rules based on Lemma 3.5 such that if for a word equation $\alpha \doteq \beta$ we have $\models \alpha \doteq \beta$ then there exists a finite sequence using $\rightarrow$ leading to the trivial word equation $\varepsilon \doteq \varepsilon$. Formally we use the reflexive transitive closure of $\rightarrow$ denoted by $\rightarrow^*$ to apply the relation $\rightarrow$ exhaustively, which is defined by 1. for all $e \in \mathtt{Pat}_A$ we have $e \rightarrow^* e$, and 2. for all $e_1 \rightarrow e_2$ and $e_2 \rightarrow e_3$ we have $e_1 \rightarrow^* e_3$. We are now ready to define the actual rules.

Let e be a word equation. The first rule eliminates equal prefixes that is if our word equation e is of the following form: $e = x\alpha \doteq x\beta$ for $x \in A \cup \mathcal{X}$ and $\alpha, \beta \in \mathtt{Pat}_A$. The rule itself is given by

$$x\alpha \doteq x\beta \rightarrow \alpha \doteq \beta. \tag{1}$$

In that sense we have $\models x\alpha \doteq x\beta$ iff $\models \alpha \doteq \beta$.

The second rule erases a variable occurring as prefix on one side of the word equation, that is if e has the form $\mathsf{x}\alpha \doteq \beta$ or $\alpha \doteq \mathsf{x}\beta$ for $\mathsf{x} \in \mathcal{X}$ and $\alpha, \beta \in \mathtt{Pat}_A$. Let $r : \mathtt{Pat}_A \rightarrow \mathtt{Pat}_A$ be morphism such that $r(\mathsf{x}) = \varepsilon$ and $r(a) = a$ for all $a \in A \cup \mathcal{X} \setminus \{\, \mathsf{x} \,\}$ then the rule is given by

$$\mathsf{x}\alpha \doteq \beta \rightarrow r(\alpha) \doteq r(\beta) \text{ and } \alpha \doteq \mathsf{x}\beta \rightarrow r(\alpha) \doteq r(\beta) \tag{2.1, 2.2}$$

We have the the following equality: there exists a solution $h \in \mathcal{H}_A$ such that $h(\mathsf{x}) = \varepsilon$ fulfilling $h \models e$ iff $\models r(\alpha) \doteq r(\beta)$.

The third rule propagates a constant occurring as prefix on one side and a variable on the other side of the word equation, that is if e has the form $a\alpha \doteq \mathsf{x}\beta$ or $\mathsf{x}\alpha \doteq a\beta$ for $\mathsf{x} \in \mathcal{X}$, $a \in A$, and $\alpha, \beta \in \mathtt{Pat}_A$. Let $r : \mathtt{Pat}_A \rightarrow \mathtt{Pat}_A$ be a morphism such that $r(\mathsf{x}) = a\mathsf{x}$ and $r(a) = a$ for all $a \in A \cup \mathcal{X} \setminus \{\, \mathsf{x} \,\}$ then the rule is given by

$$\mathsf{x}\alpha \doteq a\beta \rightarrow \mathsf{x}r(\alpha) \doteq r(\beta) \text{ and } a\alpha \doteq \mathsf{x}\beta \rightarrow r(\alpha) \doteq \mathsf{x}r(\beta) \tag{3.1, 3.2}$$

We have the the following equality: there exists a solution $h \in \mathcal{H}_A$ such that $h(\mathsf{x})[1] = a$ fulfilling $h \models \mathsf{x}\alpha \doteq a\beta$ iff $\models \mathsf{x}r(\alpha) \doteq r(\beta)$ and $h \models a\alpha \doteq \mathsf{x}\beta$ iff $\models r(\alpha) \doteq \mathsf{x}r(\beta)$.

The fourth rule makes an assumption on the prefix of our word equation if both sides starting with a variable, that is if e has the form $\mathsf{x}_1\alpha \doteq \mathsf{x}_2\beta$ for $\mathsf{x}_1, \mathsf{x}_2 \in \mathcal{X}$ and $\alpha, \beta \in \mathtt{Pat}_A$. Let $r_1, r_2, r_3 : \mathtt{Pat}_A \to \mathtt{Pat}_A$ be a morphisms such that $r_1(\mathsf{x}_1) = \mathsf{x}_2\mathsf{x}_1$, $r_2(\mathsf{x}_2) = \mathsf{x}_1\mathsf{x}_2$, $r_3(\mathsf{x}_1) = \mathsf{x}_2$, $r_1(a) = r_3(a) = a$ for all $a \in A \cup \mathcal{X} \setminus \{ \mathsf{x}_1 \}$ and $r_2(a) = a$ for all $a \in A \cup \mathcal{X} \setminus \{ \mathsf{x}_2 \}$. Then the rules are given as follows:

$$\mathsf{x}_1\alpha \doteq \mathsf{x}_2\beta \to \mathsf{x}_1 r_1(\alpha) \doteq r_1(\beta), \tag{4.1}$$

$$\mathsf{x}_1\alpha \doteq \mathsf{x}_2\beta \to r_2(\alpha) \doteq \mathsf{x}_2 r_2(\beta) \text{ and} \tag{4.2}$$

$$\mathsf{x}_1\alpha \doteq \mathsf{x}_2\beta \to r_3(\alpha) \doteq r_3(\beta). \tag{4.3}$$

We have the the following equality: there exists solutions $h_1, h_2, h_3 \in \mathcal{H}_A$ such that $h_1(\mathsf{x}_2) \in \text{prefix}(h_1(\mathsf{x}_1))$, $h_2(\mathsf{x}_1) \in \text{prefix}(h_2(\mathsf{x}_2))$, and $h_3(\mathsf{x}_2) = h_3(\mathsf{x}_1)$ fulfilling $h \models \mathsf{x}_1\alpha \doteq \mathsf{x}_2\beta$ iff $\models \mathsf{x}_1 r_1(\alpha) \doteq r_1(\beta)$, $h \models \mathsf{x}_1\alpha \doteq \mathsf{x}_2\beta$ iff $\models r_2(\alpha) \doteq \mathsf{x}_2 r_2(\beta)$, and $h \models \mathsf{x}_1\alpha \doteq \mathsf{x}_2\beta$ iff $\models r_3(\alpha) \doteq r_3(\beta)$.

As stated below each rule, transformations are preserving satisfiability. Whenever for a word equation e' we have $\models e'$ then the word equation e such that $e \to^* e'$ we have $\models e'$, meaning e is also satisfiable.

The above rules cover all word equations except for the trivially satisfiable word equation $\varepsilon \doteq \varepsilon$, and word equations of the form $a\alpha \doteq b\beta$ for all $\alpha, \beta \in \mathtt{Pat}_A$ and $a, b \in A$ such that $a \neq b$, in which we have $\not\models a\alpha \doteq b\beta$ due to mismatching prefixes. Moreover, for a word equation e_1 we have: $\models e_1$ iff there exists word equations $e_2, \ldots, e_n$ for $n \in \mathbb{N}$ such that $e_n = \varepsilon \doteq \varepsilon$ and $e_i \to e_{i+1}$ for all $i \in [n-1]_0$. Therefore, our ultimate goal is applying rules in such a way, that the length of the resulting word equation decreases in every step. The algorithm itself is now straightforward: we simply apply the reflexive transitive closure of the rules presented ($\to^*$).

To grasp the idea behind the underlying algorithm using the above rules we discuss solving a word equation within the next example.

Example 3.6. Consider the word equation

$$\alpha = a\mathsf{x}_1 ab\mathsf{x}_2 ab\mathsf{x}_1 b \doteq \mathsf{x}_2 ba\mathsf{x}_1 aabb\mathsf{x}_1 = \beta$$

also seen in Example 3.4. Evaluating the transition rules exhaustively

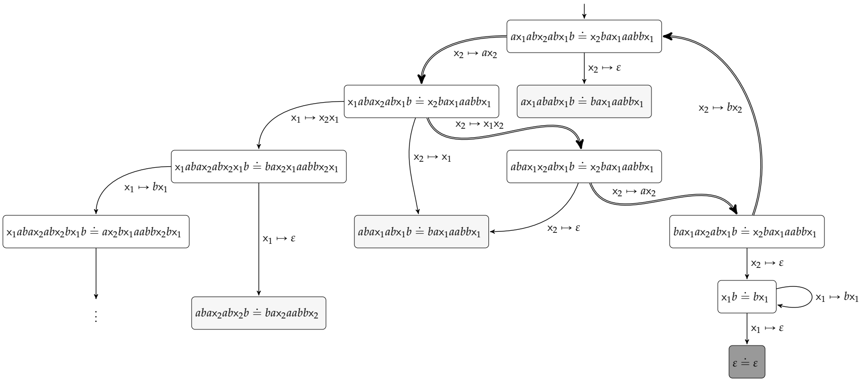

$a\mathtt{x}_1ab\mathtt{x}_2ab\mathtt{x}_1b \doteq \mathtt{x}_2ba\mathtt{x}_1aabb\mathtt{x}_1$
$\mathtt{x}_2 \mapsto a\mathtt{x}_2$
$\mathtt{x}_2 \mapsto \varepsilon$
$\mathtt{x}_2 \mapsto b\mathtt{x}_2$
$\mathtt{x}_1aba\mathtt{x}_2ab\mathtt{x}_1b \doteq \mathtt{x}_2ba\mathtt{x}_1aabb\mathtt{x}_1$
$a\mathtt{x}_1abab\mathtt{x}_1b \doteq ba\mathtt{x}_1aabb\mathtt{x}_1$
$\mathtt{x}_1 \mapsto \mathtt{x}_2\mathtt{x}_1$
$\mathtt{x}_2 \mapsto \mathtt{x}_1\mathtt{x}_2$
$\mathtt{x}_2 \mapsto \mathtt{x}_1$
$\mathtt{x}_1aba\mathtt{x}_2ab\mathtt{x}_2\mathtt{x}_1b \doteq ba\mathtt{x}_2\mathtt{x}_1aabb\mathtt{x}_2\mathtt{x}_1$
$aba\mathtt{x}_1\mathtt{x}_2ab\mathtt{x}_1b \doteq \mathtt{x}_2ba\mathtt{x}_1aabb\mathtt{x}_1$
$\mathtt{x}_1 \mapsto b\mathtt{x}_1$
$\mathtt{x}_2 \mapsto a\mathtt{x}_2$
$\mathtt{x}_1aba\mathtt{x}_2ab\mathtt{x}_2b\mathtt{x}_1b \doteq a\mathtt{x}_2b\mathtt{x}_1aabb\mathtt{x}_2b\mathtt{x}_1$
$\mathtt{x}_1 \mapsto \varepsilon$
$aba\mathtt{x}_1ab\mathtt{x}_1b \doteq ba\mathtt{x}_1aabb\mathtt{x}_1$
$\mathtt{x}_2 \mapsto \varepsilon$
$ba\mathtt{x}_1a\mathtt{x}_2ab\mathtt{x}_1b \doteq \mathtt{x}_2ba\mathtt{x}_1aabb\mathtt{x}_1$
$\mathtt{x}_2 \mapsto \varepsilon$
$\mathtt{x}_1b \doteq b\mathtt{x}_1$
$\mathtt{x}_1 \mapsto b\mathtt{x}_1$
$aba\mathtt{x}_2ab\mathtt{x}_2b \doteq ba\mathtt{x}_2aabb\mathtt{x}_2$
$\mathtt{x}_1 \mapsto \varepsilon$
$\varepsilon \doteq \varepsilon$

Figure 3.8. Applying the transformation rules until a solution is found

leads to the huge graph. Usually, in practice applying the rules is stopped whenever a solution is found or – in case of an unsatisfiable word equation – all leafs are marked as unsatisfiable. We depict the graph resulting by applying our rules to the word equation $\alpha \doteq \beta$ in Figure 3.8. Note, we omit extra nodes for erasing prefixes (Rule 1) to ease readability. Nodes in light grey indicate unsatisfiable word equations, whereas dark grey indicates a satisfiable one. Each node is annotated with the corresponding rule leading there. We can now obtain the solution by traversing the graph starting at our satisfiable dark grey node labelled $\varepsilon = \varepsilon$ to our initial node being labelled by the word equation we try to solve. It reveals the assignment $h = \{ \mathtt{x}_1 \mapsto \varepsilon, \mathtt{x}_2 \mapsto aa \}$ which is indeed a solution. This small extraction of the graph reveals infinitely many solutions since it contains self edges (i.e the node $\mathtt{x}_1 b \doteq b\mathtt{x}_1$) and a loop (highlighted with doubled arrows in Figure 3.8). We immediately get $h(\mathtt{x}_1) \in \{ b \}^*$. Furthermore, following the loop we get $h(\mathtt{x}_2) \in \{ a\, h(\mathtt{x}_1)\, a(ba\, h(\mathtt{x}_1)\, a)^* \}$ for any $h(\mathtt{x}_1) \in \{ b \}^*$. This example shows that the graph induced by the above mentioned rewrite rules allows analysing more than just a single solution.

For specific classes of word equations the termination of an exhaustive application of the given rules is known. Also many word equations encountered in practice are of a particular form. Well studied forms, where termination is known are quadratic and regular word equations [89]. A word equation $\alpha \doteq \beta \in \texttt{Pat}_A$ is called quadratic if $|\alpha_{\mathsf{x}}| + |\beta_{\mathsf{x}}| \leqslant 2$ for all $\mathsf{x} \in \mathcal{X}$ and regular if a variable occurs at most once per side. For quadratic word equations, the graphs obtained by applying the rules of our transformation system and in particular the rules derived from Levi's lemma contain some natural structure. In particular, it is well-known that the length of the equations never increases when following an edge, meaning the graph is finite. This is of course not necessarily true for word equations in general. It means that the graphs can be decomposed into individual layers corresponding to word equations of a given length. Each layer consisting of possibly several connected components. A path from the original equation to a trivial one will necessarily travel through each layer in order. For regular equations (each variable occurs at most once per side), the connected components of each layer will each be strongly connected, meaning that all equations in that component are equisatisfiable. This suggests a probable high number of inconsequential choices when searching the graph for a path to a trivial equation.

Empirical observations also seem to indicate that, at least for regular word equations with less than 10 variables, the diameter of the graphs is low, and the graphs are typically very well connected. By considering certain regular equations, it can also be observed that applying Levi's lemma to both prefixes and suffixes of the equation has the effect of increasing the out-degree of vertices, and the number of vertices, without necessarily having any positive effect on the diameter, and thus it might be expected that applying the transformations only in one direction is more efficient. This is not restricting the set of possible solutions. Consider a word equation $\alpha \doteq \beta \in \texttt{Pat}_A$ with a solution h, then $h^{\mathsf{R}} = \left\{ X \mapsto h(X)^{\mathsf{R}} \right\}$ is a solution to the reversed word equation $\alpha^{\mathsf{R}} \doteq \beta^{\mathsf{R}}$. However, in practice, due to our empirical studies, it is mostly more efficient to apply both, prefix and suffix rules.

Unfortunately, the useful structures found in quadratic and regular equations are not necessarily present any more in the general case, where the length of the equation will often increase when following an edge. This makes it much harder to predict the structure of the graph, and to determine whether similar symmetries and high connectivity are present.

Of course, if the graph is not neccessarily finite, then neither is the diameter of the graph. On the other hand, there are some reasonable guesses, or assumptions about the graph which might provide an indication as to how effective performance may be achieved when searching for paths in these graphs. Firstly, as the equations become longer, it is possible that more conflicts or contradictions may arise, meaning that this area of the graph is not worth exploring, or similarly, that a simpler solution will exist, meaning a shorter path to a trivial equation will exist elsewhere. Similarly, for variables with a similar number of occurrences on both sides of the equation, it is not unreasonable to expect that often solutions exist for which each of the variables is (slightly) longer than the other. Thus there may also be a number of equisatisfiability symmetries in the general graphs, although more restricted and harder to pinpoint than in the graphs of regular equations. Finally, since many practical examples will either be quadratic, respectively regular, or mostly quadratic, respectively regular, it is also possible that often, large parts of the graph will be similar to the graphs of quadratic, respectively regular equations, and thus display similarly desirable properties.

This introduction to solving strategies closes the chapter covering the application based embedding of string constraints. The procedures we will dicuss in the next chapter build on top of these algorithms, following our goal to design specialised methods to solving string constraints.

Chapter 4

Different Techniques to Solving String Constraints

"The one-way culture must be stopped."

THE MOVEMENT

In recent years, word equations have gained attention from the formal verification and security community, because word equations naturally occur during symbolic execution of high-level languages. We covered basic approaches to solving word equations and also the most prominent tools in the previous section. As we observed the approaches to solving a particular formula varies heavily. Nevertheless, all tools try to achieve the same goal: determining whether a word equation (possibly involving extra constraints) is satisfiable or not. Within this chapter we enrich this landscape by presenting novel approaches to target this area. The first section of this chapter presents a hybrid automata- and unfolding-based approach encoding string constraints into a propositional logic formula. The second section introduces a word-based approach being derived from Levi's lemma. In the third section we identify several sub-theories of string constraints involving regular membership predicates based on real-world benchmarks and prove decidability for some of the resulting theories. Furthermore, we use our proofs to extend Z3STR3's algorithm for solving regular membership predicates.

In particular, because solving string formulae is believed to be hard in general, solver designers have come up with a diverse set of practical algorithms that incorporate a variety of tradeoffs. Some of these methods work well for pure word equations, but not so well whenever adding additional constraints. This diversity of algorithms for solving string constraints immediately leads to the goal of understanding each of them in a better way what we try to achieve with our contributions.

This chapter is based on work being published at RP 2019 [40], FormaliSE 2020 [42], CAV 2021 [24], and WORDS 2021 [22].

4.1 Encoding String Constraints into SAT

This section presents an approach to solving the theory of word equation with linear length constraints $\mathcal{A}_l^{\doteq}$ which was discussed in Section 2.3.1. Keep in mind that deciding whether a word equation involving a linear length constraint still remains a major open problem.

The key idea of our procedure is an unfolding-based approach encoding word equations and length constraint into a single propositional logic formula in order to use a SAT solver to obtain a solution. In particular, we are guessing the maximal length of variables and encode a variation of the *filling the positions* method (presented in Section 3.5.1) into an automata-construction, thereby reducing the search for a solution to a reachability question of this automata. Preliminary experiments with a pure automata-reachability-based approach revealed however, that this does not scale, even for small word equations.

We propose an algorithm solving the bounded word equation problem. In this section, we describe an approach for transforming a word equation $\alpha \doteq \beta$ for patterns $\alpha, \beta \in \mathtt{Pat}_A$ into a finite automaton, which is then directly encodable into propositional logic. Consider an equation $\alpha \doteq \beta$. If it is satisfiable then there exists a substitution $h \in \mathcal{H}_A$ such that $h(\alpha) = h(\beta)$ as well as a function $\mathcal{L} : \mathcal{X} \to \mathbb{N}$ which assigns a positive integer length ℓ_{x} to each variable $\mathsf{x} \in \mathcal{X}$. It is rather easy to see that it is enough to know the precise length of the image of each variable in order to see whether the word equation has a solution or not because knowning exact length of each variable lets us obtain a solution by using the technique presented in Section 3.5.1. As such, finding the function $\mathcal{L}$ is as hard as solving the word equation.

Coming back to formal background presented for word equations in

Section 2.2.1, this enriched setting changes our vocabulary to

$$\check{\mathcal{W}} = \{ \cdot//2, \mathcal{L}//1, \mathsf{len}//1, \leqslant /2, \dot{\varepsilon} \},$$

adding a new function $\mathcal{L}$ taking care of our bounds and a function len known from Section 2.3.1, the combination of word equations and length constraints allows to reason about the length of a solution for a variable. To express the constraint we also need a comparison relation $\leqslant$. The Σ_1 fragment of $\mathsf{FO}(\check{\mathcal{W}})$ therefore additionally contains the axiom

$$\forall \mathsf{x}_1 \,.\, \mathcal{L}(\mathsf{x}_1) \leqslant \mathsf{len}(\mathsf{x}_1)$$

accommodating the newly added functions. Additionally, we require $\leqslant$ to be a total order. Therefore, we add the corresponding axioms given in Section 2.2.2. Consider the many-sorted $\check{\mathcal{W}}$-structure

$$\mathcal{A}_{\check{\mathrm{W}}} = \left\{ A^*, \mathbb{N}, \cdot^A, \dot{\varepsilon}^A, \leqslant^{\mathbb{N}}, \mathcal{L}^{\mathcal{X}\to\mathbb{N}}, \mathsf{len}^{\mathcal{X}\to\mathbb{N}} \right\},$$

having the semantics given in Section 2.3.1, and $\mathcal{L}^{\mathcal{X}\to\mathbb{N}}$ is an arbitrary, total mapping of our variables to a positive integer. This first order logic fragment is given for the sake of completeness. Following the shape of this work and to ease readability, we will usually omit the instantiation of the structure.

The rest of this section is structured as follows: we describe our approach by first solving bounded word equations, and secondly, we discuss a minor change, that allows solving word equations with linear constraints as discussed in Section 2.3.1. Before concluding, we take a brief look at closely related work and provide details on our experimental implementation of the presented approach.

4.1.1 Solving Bounded Word Equation

Recall that a bounded word equation consists of a word equation $\alpha \doteq \beta$ along with a set of equations $\{\mathsf{len}(\mathsf{x}) \leqslant b_{\mathsf{x}} | \mathsf{x} \in \mathcal{X}\}$ providing upper bounds for the solution of each variable $\mathsf{x} \in \mathcal{X}$. In our approach we use these bounds to create a finite automaton which has an accepting run if and only if the bounded word equation is satisfiable.

Before the actual automata construction can be presented, we need some convenient transformations of the word equation itself.

Definition 4.1. Let $\mathcal{L} : \mathcal{X} \to \mathbb{N}$ be a length bound function. Let b_{x} denote the length bound for a variable x with respect to $\mathcal{L}$, i.e. $\mathcal{L}(\mathsf{x}) = b_{\mathsf{x}}$. We call the set $\check{\mathcal{X}} = \left\{ \mathsf{x}^{(i)} \;\middle|\; \mathsf{x} \in \mathcal{X}, i \in [b_{\mathsf{x}}] \right\}$ *filled variables* alphabet, where for a variable x we call each $\mathsf{x}^{(i)}$ for $i \in [b_{\mathsf{x}}]$ a *filled variable*.
Let $\mathtt{P\check{a}t}_A = \left(A \cup \check{\mathcal{X}}\right)^*$ denote the set of *filled patterns*.

Note, whenever we state b_{x} for a variable $\mathsf{x} \in \mathcal{X}$ we refer to x's corresponding length bound indicated by a length bound function $\mathcal{L} : \mathcal{X} \to \mathbb{N}$.

Conveniently, when considering a word equation $\alpha \doteq \beta$ for $\alpha, \beta \in \mathtt{Pat}_A$ we obtain the corresponding word equation over its filled pattern by simply mapping each variable to it corresponding sequence of filled variables, formally defined as follows.

Definition 4.2. Let $\alpha \in \mathtt{Pat}_A$ and $\mathcal{L} : \mathcal{X} \to \mathbb{N}$ be a length bound function. We obtain the filled pattern by

$$\check{\alpha} = \begin{cases} \varepsilon & \text{if } \alpha = \varepsilon, \\ a \cdot \check{\alpha}[2:] & \text{if } a = \alpha[1] \in A, \\ \mathsf{x}^{(0)} \ldots \mathsf{x}^{(b_{\mathsf{x}}-1)} \cdot \check{\alpha}[2:] & \text{if } \mathsf{x} = \alpha[1] \in \mathcal{X} \text{ and } \mathcal{L}(\mathsf{x}) = b_{\mathsf{x}}. \end{cases}$$

To be consistent with respect to the original word equation we restrict filled variables to only be substituted by either a single letter or the empty word.

Definition 4.3. Let $\lambda \notin A$ be a fresh symbol and set $A_\lambda = A \cup \{ \lambda \}$, $h \in \mathcal{H}_A$ and $\mathcal{L} : \mathcal{X} \to \mathbb{N}$ be a length bound function. We canonically define the induced substitution $\check{h} : \mathtt{P\check{a}t}_A \to A_\lambda$ for filled patterns by

1. $\check{h}(a) = h(a)$ for all $a \in A$,
2. $\check{h}\left(\mathsf{x}^{(i)}\right) = h(\mathsf{x})[i]$ for all $\mathsf{x}^{(i)} \in \check{\mathcal{X}}$ and $i \in \left[|\check{h}(\mathsf{x})|\right]$, and
3. $\check{h}\left(\mathsf{x}^{(j)}\right) = \lambda$ for all $\mathsf{x}^{(j)} \in \check{\mathcal{X}}$ and $|\check{h}(\mathsf{x})| \leqslant j < \mathcal{L}(\mathsf{x})$.

Here, λ is a new symbol to indicate an unused position at the end of a filled variable. Note that the substitution of a single filled variable always maps to exactly one character from A_λ, and, as soon as we discover $\check{h}\left(\mathsf{x}^{(j)}\right) = \lambda$ for $j \in [b_X]$ it also holds that $\check{h}\left(\mathsf{x}^{(i)}\right) = \lambda$ for all $j \leqslant i < b_X$. In a sense, the new element λ behaves in the same way as the neutral element of the monoid over A^*, being actually a place holder for the element ε. In the other direction, if we have found a satisfying filled substitution to our word equation, the two filled patterns obtained from the left hand side and the right hand side of an equation, respectively, we can transform it to a substitution for our original word equation by defining our solution as the concatenation of our filled substitution in which each occurrence of λ is simply removed.

Definition 4.4. Let $\check{h} : \check{\mathtt{Pat}}_A \to A_\lambda$ be a filled substitution. We obtain the substitution $h \in \mathcal{H}_A$ for each variable $\mathsf{x} \in \mathcal{X}$ having a length bound b_x by

$$h(\mathsf{x}) = \check{h}(\mathsf{x}^{(0)}) \ldots \check{h}(\mathsf{x}^{(i-1)}),$$

for $i \in [b_\mathsf{x}]$ such that $\check{h}(\mathsf{x}^{(j)}) = \lambda$ for all $i \leqslant j < b_\mathsf{x}$ and $\check{h}(\mathsf{x}^{(i-1)}) \neq \lambda$.

Our goal is now to build an automaton which calculates a suitable substitution for a given equation. During the calculation there are situations where a substitution does not form a total function. To extend a partial substitution $h \in \mathcal{H}_A$ we define for $\mathsf{x} \in \mathcal{X}$ and $a \in A$ the notation

$$h\left[\frac{\mathsf{x}}{b}\right] = \begin{cases} h \cup \{\, \mathsf{x} \mapsto b \,\} & \text{if } \mathsf{x} \notin \mathsf{dom}(h), \\ h & \text{otherwise.} \end{cases}$$

This definition can be naturally applied to filled substitutions.

In order to fully define our automaton we need a congruence relation which sets variables and letters in relation whenever their substitution with respect to a partial substitution $\check{h}$ is equal or undefined. Formally this relation is expressed as follows.

Definition 4.5. We define $\overset{\check{h}}{\sim} \subseteq \check{\mathtt{Pat}}_{A_\lambda} \times \check{\mathtt{Pat}}_{A_\lambda}$ by

$$\alpha \overset{\check{h}}{\sim} \beta \text{ iff } \check{h}(\alpha) = \check{h}(\beta) \text{ or } \check{h}(\beta) \notin A_\lambda \text{ or } \check{h}(\alpha) \notin A_\lambda,$$

for all $\alpha, \beta \in \check{\mathtt{Pat}}_{A_\lambda} \cup \{\, \lambda \,\}$ and $\check{h} : \check{\mathtt{Pat}}_{A_\lambda} \to A_\lambda$.

We are now ready to define our automaton which encodes a particular word equation.

Definition 4.6. For a word equation $\alpha \doteq \beta$ for $\alpha, \beta \in \mathtt{Pat}_A$ and a mapping $\mathcal{L} : \mathcal{X} \to \mathbb{N}$ defining the bounds $\mathcal{L}(\mathsf{x}) = b_{\mathsf{x}}$, we define the *equation automaton* $A(\alpha \doteq \beta, \mathcal{L}) = (Q, \delta, I, F)$ where $Q = \big([|\check{\alpha}| + 1] \times [|\check{\beta}| + 1]\big) \times \big(\check{\mathtt{Pat}}_{A_\lambda} \rightharpoonup A_\lambda\big)$ is a set of states consisting of two integers which indicate the position inside the two words $\check{\alpha}$ and $\check{\beta}$ and a partial substitution, the transition function $\delta : Q \times A_\lambda \to Q$ defined by

$$\delta\left(\left(\left(i,j\right),h\right),a\right) = \begin{cases} \left(\left(i+1,j+1\right),h\left[\frac{\check{\alpha}[i]}{a}\right]\left[\frac{\check{\beta}[j]}{a}\right]\right) & \text{if } \check{\alpha}[i] \overset{\check{h}}{\sim} \check{\beta}[j] \overset{\check{h}}{\sim} a, \\ \left(\left(i+1,j\right),h\left[\frac{\check{\alpha}[i]}{\lambda}\right]\right) & \text{if } \check{\alpha}[i] \overset{\check{h}}{\sim} \lambda = a, \\ \left(\left(i,j+1\right),h\left[\frac{\check{\beta}[j]}{\lambda}\right]\right) & \text{if } \check{\beta}[j] \overset{\check{h}}{\sim} \lambda = a, \end{cases}$$

an initial state $I = \left(\left(0,0\right), \{\, a \mapsto a \mid a \in \Sigma_\lambda \,\}\right)$ and the set of final states $F = \left\{ \left(\left(i,j\right),\check{h}\right) \,\middle|\, i = |\check{\alpha}|, j = |\check{\beta}| \right\}$.

The definition of the final states assumes that all variables in $\mathcal{X}$ are used inside the given equation meaning $\mathtt{vars}\,(\alpha) \cup \mathtt{vars}\,(\beta) = \mathcal{X}$. Therefore, throughout this section we consider $\mathcal{X}$ being finite. We can of course assume this without restriction. The state space of our automaton is finite since the filled substitution $\check{h}$ maps each input to exactly one character in A_λ. The automaton is nondeterministic, as the three choices we have for a transition are not necessarily mutually exclusive.

As an addition to Definition 4.6, we introduce the notion of *location* as a pair of integers (i, j) corresponding to two positions inside the two pattern $\check{\alpha}$ and $\check{\beta}$. A location (i, j) can also be seen as the set of states of the form $((i, j), h)$ for all possible partial substitutions h.

A run of the above nondeterministic automaton constructs a partial substitution for the given equation which is extended with each change of state. The equation has a solution if one of the accepting states $(|\check{\alpha}|, |\check{\beta}|, h)$, where $h \in \mathcal{H}_A$ is a total substitution, is reachable, because the automaton simulates a walk through our input equation left to right, with all its positions filled in a coherent way.

Example 4.7. Consider the equation $\alpha \doteq \beta$ for $\alpha = a\mathsf{x}_3\mathsf{x}_1 b, \beta = a\mathsf{x}_1 a\mathsf{x}_2 \in \mathtt{Pat}_A$. We choose the bounds $b_{\mathsf{x}_1} = b_{\mathsf{x}_2} = b_{\mathsf{x}_3} = 1$. This will give us the words $\check{\alpha} = a{\mathsf{x}_3}^{(0)}{\mathsf{x}_1}^{(0)}b$ and $\check{\beta} = a{\mathsf{x}_1}^{(0)}a{\mathsf{x}_2}^{(0)}$. Figure 4.9 visualizes the corresponding automaton. A run starting with the initial substitution $h_i =$

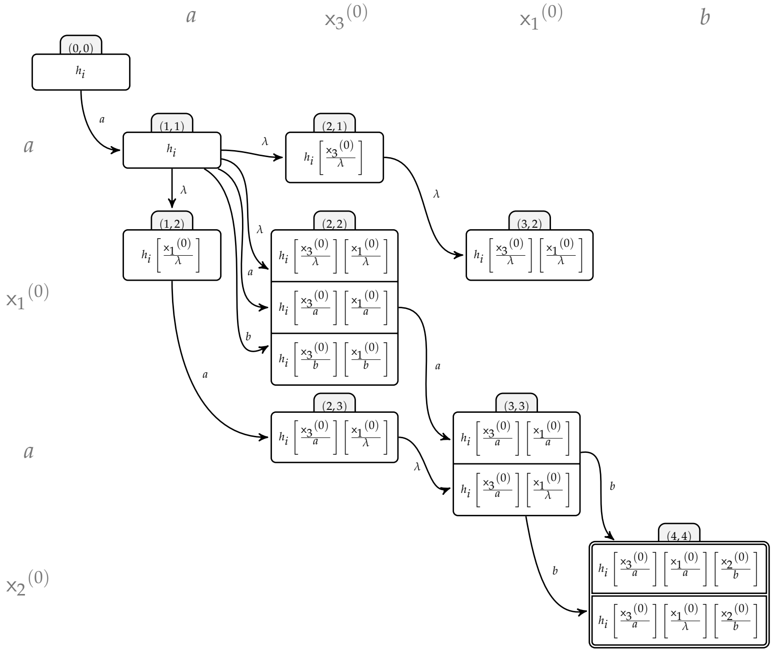

a
$\mathsf{x}_3{}^{(0)}$
$\mathsf{x}_1{}^{(0)}$
b
a
$\mathsf{x}_1{}^{(0)}$
a
$\mathsf{x}_2{}^{(0)}$
(0,0)
h_i
(1,1)
h_i
(2,1)
$h_i \left[\frac{\mathsf{x}_3{}^{(0)}}{\lambda}\right]$
(1,2)
$h_i \left[\frac{\mathsf{x}_1{}^{(0)}}{\lambda}\right]$
(2,2)
$h_i \left[\frac{\mathsf{x}_3{}^{(0)}}{\lambda}\right] \left[\frac{\mathsf{x}_1{}^{(0)}}{\lambda}\right]$
$h_i \left[\frac{\mathsf{x}_3{}^{(0)}}{a}\right] \left[\frac{\mathsf{x}_1{}^{(0)}}{a}\right]$
$h_i \left[\frac{\mathsf{x}_3{}^{(0)}}{b}\right] \left[\frac{\mathsf{x}_1{}^{(0)}}{b}\right]$
(3,2)
$h_i \left[\frac{\mathsf{x}_3{}^{(0)}}{\lambda}\right] \left[\frac{\mathsf{x}_1{}^{(0)}}{\lambda}\right]$
(2,3)
$h_i \left[\frac{\mathsf{x}_3{}^{(0)}}{a}\right] \left[\frac{\mathsf{x}_1{}^{(0)}}{\lambda}\right]$
(3,3)
$h_i \left[\frac{\mathsf{x}_3{}^{(0)}}{a}\right] \left[\frac{\mathsf{x}_1{}^{(0)}}{a}\right]$
$h_i \left[\frac{\mathsf{x}_3{}^{(0)}}{a}\right] \left[\frac{\mathsf{x}_1{}^{(0)}}{\lambda}\right]$
(4,4)
$h_i \left[\frac{\mathsf{x}_3{}^{(0)}}{a}\right] \left[\frac{\mathsf{x}_1{}^{(0)}}{a}\right] \left[\frac{\mathsf{x}_2{}^{(0)}}{b}\right]$
$h_i \left[\frac{\mathsf{x}_3{}^{(0)}}{a}\right] \left[\frac{\mathsf{x}_1{}^{(0)}}{\lambda}\right] \left[\frac{\mathsf{x}_2{}^{(0)}}{b}\right]$
a
λ
λ
λ
λ
a
b
a
a
λ
b
b

Figure 4.1. Automaton for the word equation $a\mathsf{x}_3\mathsf{x}_1 b \doteq a\mathsf{x}_1 a\mathsf{x}_2$, with the states grouped according to their locations. Only reachable states are shown.

$\{\, a \mapsto a \mid a \in A_\lambda \,\}$ reaching one of the final states gives us a solution to the equation. In this example we get the substitutions $\mathsf{x}_3 \mapsto a, \mathsf{x}_1 \mapsto a, \mathsf{x}_2 \mapsto b$ and $\mathsf{x}_3 \mapsto a, \mathsf{x}_1 \mapsto \varepsilon, \mathsf{x}_2 \mapsto b$.

As stated, the construction of the equation automaton clearly translates the search for a suitable solution into a reachability problem. The correct behaviour of this translation is implied by the following theorem.

Theorem 4.8. *Given a bounded word equation $\alpha \doteq \beta$ for $\alpha, \beta \in \mathtt{Pat}_A$, with bounds $\mathcal{L} : \mathcal{X} \to \mathbb{N}$, then in the automaton $A(\alpha \doteq \beta, \mathcal{L})$ an accepting state is reachable if and only if there exists h such that $h \models \alpha \doteq \beta$ and $|h(\mathsf{x})| \leqslant \mathcal{L}(\mathsf{x})$ for all $\mathsf{x} \in \mathcal{X}$.*

In order to argue about the correctness of our construction it is enough to note that the transitions of the automaton from a state $((i,j),h)$ increase at least one of the components i or j, and while doing so, they align two identical symbols from the left hand side and right hand side, respectively, of the filled equation, either by assigning a value to one or both of them, or by processing an identical symbol in both sides. Clearly, a final state is reached if and only if an assignment that makes the two sides equal is reached. A formal proof of the above theorem can be done easily by induction, and is left to the reader.

In the next section we will encode the automaton and a corresponding run into propositional logic, which allows us using a SAT solver to obtain a solution to a bounded word equation.

4.1.2 SAT Encoding

We now encode the solving process into propositional logic. For that we impose an ordering on the finite alphabets $A = \{ a_0, \ldots, a_{n-1} \}$ and $\mathcal{X} = \{ \mathsf{x}_0, \ldots, \mathsf{x}_{m-1} \}$ for $n, m \in \mathbb{N}$. Using the upper bounds given for all variables $\mathsf{x} \in \mathcal{X}$, we create the filled variables alphabet $\check{\mathcal{X}}$.

To improve readability and not confuse with variables within our word equations let $\mathcal{X}_{\mathbb{B}}$ denote the finite set of propositional logic formulae and $\mathcal{B} = 2^{\mathcal{X}_{\mathbb{B}} \times \{ 0,1 \}}$ denote the set of all propositional logic assignments. Further, we create the propositional logic variables

$$\mathsf{K}^{a}_{\mathsf{x}^{(i)}} \in \mathcal{X}_{\mathbb{B}},$$

for all $\mathsf{x}^{(i)} \in \check{\mathcal{X}}$, $a \in A_{\lambda}$ and $i \in [b_{\mathsf{x}}]$. Intuitively, we want to construct our formula such that an assignment $s \in \mathcal{B}$ sets $\mathsf{K}^{a}_{\mathsf{x}^{(i)}}$ to 1, if the solution of the word equation, which corresponds to the assignment s, is such that at position i of the variable X an a is found, meaning $\check{h}\left(X^{(i)}\right) = a$.

To make sure $\mathsf{K}^{a}_{X^{(i)}}$ is set to 1 for exactly one $a \in A_{\lambda}$ we define the clause

$$\bigvee_{a \in A_{\lambda}} \mathsf{K}^{a}_{\mathsf{x}^{(i)}}$$

which needs to be assigned true, as well as the constraints

$$\mathsf{K}^{a}_{\mathsf{x}^{(i)}} \rightarrow \neg \mathsf{K}^{b}_{\mathsf{x}^{(i)}},$$

for all $a, b \in A_{\lambda}, \mathsf{x} \in \mathcal{X}, i \in [b_{\mathsf{x}}]$ where $a \neq b$, which also all need to be true.

To match letters we add the variables

$$\mathsf{C}_{a,a} \leftrightarrow \top \text{ and } \mathsf{C}_{a,b} \leftrightarrow \bot$$

for all $a, b \in \Sigma_\lambda$ with $a \neq b$. As such, the actual encoding of our equation can be defined as follows: for $w \in \{ \check{\alpha}, \check{\beta} \}$ and each position i of w and letter $a \in A_\lambda$ we introduce a variable which is true if and only if $w[i]$ will correspond to an a in the solution of the word equation. More precisely, we make a distinction between constant letters and variable positions and define:

$$\mathsf{word}^a_{w[i]} \leftrightarrow \begin{cases} \mathsf{C}_{w[i],a} & \text{if } w[i] \in A_\lambda, \\ \mathsf{K}^a_{w[i]} & \text{if } w[i] \in \check{\mathcal{X}}. \end{cases}$$

The equality of two characters corresponding to position i in α and, respectively, j in β, is encoded by introducing a propositional logic variable $\mathsf{wm}_{i,j} \in \mathcal{X}_\mathbb{B}$ such that

$$\mathsf{wm}_{i,j} \leftrightarrow \bigvee_{a \in \Sigma_\lambda} \mathsf{word}^a_{\alpha[i]} \wedge \mathsf{word}^a_{\beta[j]}$$

for appropriate $i \in [|\check{\alpha}|]$, $j \in [|\check{\beta}|]$.

As seen in Definition 4.6 we process both sides of the equation simultaneously, from left to right. Based on this setup, we assign a propositional logic variable to each location of the automaton. As such, for a given equation $\alpha \doteq \beta$ we create $n \cdot m = (|\check{\alpha}| + 1) \cdot \big(|\check{\beta}| + 1\big)$ many propositional logic variables

$$\mathsf{S}_{i,j}$$

for $i \in [n]$ and $j \in [m]$. The location $(0, 0)$ is our initial location and $(|\check{\alpha}|, |\check{\beta}|)$ our accepting location. The goal is to find a path between those two locations, or, alternatively, a satisfying assignment $s \in \mathcal{B}$, which sets the variables corresponding to these locations to 1. Every path between the location $(0, 0)$ and another location corresponds to matching prefixes of α and β under proper substitutions. We will call locations where an assignment s sets a variable $\mathsf{S}_{i,j}$ to 1, active locations. Our transitions are now defined by a set of constraints. We fix $i \in [n]$ and $j \in [m]$ in the following. The constraints are given as follows: The first constraint ensures that every active location has at least one active successor.

$$\mathsf{S}_{i,j} \rightarrow \mathsf{S}_{i+1,j} \vee \mathsf{S}_{i,j+1} \vee \mathsf{S}_{i+1,j+1}. \tag{4.1}$$

The next three constraints ensure the validity of the paths we use: from a location we can only proceed to exactly one other location, in order to find a satisfying assignment; therefore we disallow simultaneous steps in multiple directions.

$$\left(\mathsf{S}_{i,j} \wedge \mathsf{S}_{i,j+1}\right) \rightarrow \left(\neg\mathsf{S}_{i+1,j+1} \wedge \neg\mathsf{S}_{i+1,j}\right) \tag{4.2}$$

$$\left(\mathsf{S}_{i,j} \wedge \mathsf{S}_{i+1,j}\right) \rightarrow \left(\neg\mathsf{S}_{i+1,j+1} \wedge \neg\mathsf{S}_{i,j+1}\right) \tag{4.3}$$

$$\left(\mathsf{S}_{i,j} \wedge \mathsf{S}_{i+1,j+1}\right) \rightarrow \left(\neg\mathsf{S}_{i,j+1} \wedge \neg\mathsf{S}_{i+1,j}\right) \tag{4.4}$$

We also forbid using an λ-transition whenever there is another possibility of moving forward

$$\mathsf{S}_{i,j} \wedge \neg\mathsf{word}^{\lambda}_{\alpha[i]} \rightarrow \neg\mathsf{S}_{i+1,j} \text{ and } \mathsf{S}_{i,j} \wedge \mathsf{word}^{\lambda}_{\alpha[i]} \wedge \neg\mathsf{word}^{\lambda}_{\beta[j]} \rightarrow \mathsf{S}_{i+1,j}, \tag{4.5}$$

$$\mathsf{S}_{i,j} \wedge \neg\mathsf{word}^{\lambda}_{\beta[j]} \rightarrow \neg\mathsf{S}_{i,j+1} \text{ and } \mathsf{S}_{i,j} \wedge \neg\mathsf{word}^{\lambda}_{\alpha[i]} \wedge \mathsf{word}^{\lambda}_{\beta[j]} \rightarrow \mathsf{S}_{i,j+1}. \tag{4.6}$$

In the same manner we guide the path for two matching λ. This part is especially important for finding substitutions which are smaller than the given bounds. The idea applies in the same way for matching letters, whose encoding is given in (4.8).

$$\mathsf{S}_{i,j} \wedge \mathsf{word}^{\lambda}_{\alpha[i]} \wedge \mathsf{word}^{\lambda}_{\beta[j]} \rightarrow \mathsf{S}_{i+1,j+1} \tag{4.7}$$

$$\mathsf{S}_{i,j} \wedge \mathsf{S}_{i+1,j+1} \rightarrow \mathsf{wm}_{i,j} \tag{4.8}$$

The actual transitions which are possible from one state to another are encoded by using our propositional logic words match variables $\mathsf{wm}_{i,j}$ which are true for matching positions in the two sides of the equation

$$\begin{aligned}\mathsf{S}_{i,j} \leftrightarrow \quad & \left(\mathsf{S}_{i-1,j-1} \wedge \mathsf{wm}_{i-1,j-1}\right)\\ \vee\, & \left(\mathsf{S}_{i,j-1} \wedge \neg\mathsf{wm}_{i,j-1}\right)\\ \vee\, & \left(\mathsf{S}_{i-1,j} \wedge \neg\mathsf{wm}_{i-1,j}\right).\end{aligned} \tag{4.9}$$

This constraint allows us to move forward in both words if there was a match of two letters in the previous location. When the transitions are pictured as movements in the plane, this corresponds to a diagonal move. A horizontal or vertical move corresponds to a match with the empty word. The last constraint ensures a valid predecessor. This is supposed to help the solver in deciding the satisfiability of the obtained formula, i.e., to

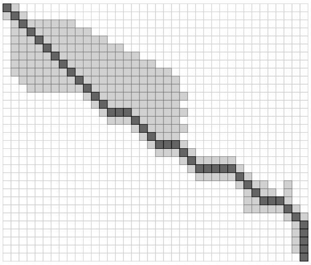


Figure 4.2. Solver computation on $\mathsf{x}_1 a \mathsf{x}_1 b \mathsf{x}_2 b \mathsf{x}_3 \doteq a \mathsf{x}_1 \mathsf{x}_2 \mathsf{x}_2 b \mathsf{x}_3 \mathsf{x}_3 baa$

guide the search in an efficient way. It can be seen as a local optimization step.

$$\mathsf{S}_{i+1,j+1} \rightarrow \quad \mathsf{S}_{i,j} \vee \mathsf{S}_{i+1,j} \vee \mathsf{S}_{i,j+1}. \tag{4.10}$$

The final formula is the conjunction of all constraints defined above. This formula is true iff location (n,m) is reachable from location $(0,0)$, and this is true iff the given word equation is satisfiable w.r.t. the given length bounds.

Lemma 4.9. *Let $\alpha \doteq \beta$ be a word equation, $\mathcal{L} : \mathcal{X} \rightarrow \mathbb{N}$ be the function giving the bounds for the word equation variables, and φ the corresponding formula consisting of the conjunction (4.1) - (4.10) and the earlier defined constraints in this section, then $\varphi \wedge \mathsf{S}_{0,0} \wedge \mathsf{S}_{|\check{\alpha}|,|\check{\beta}|}$ has a satisfying assignment if and only if in the automaton $A(\alpha \doteq \beta, \mathcal{L})$ an accepting state is reachable.*

Example 4.10. Consider the word equation $\alpha \doteq \beta$ where $\alpha = \mathsf{x}_1 a \mathsf{x}_1 b \mathsf{x}_2 b \mathsf{x}_3$ and $\beta = a \mathsf{x}_1 \mathsf{x}_2 \mathsf{x}_2 b \mathsf{x}_3 \mathsf{x}_3 baa \in \mathtt{Pat}_A$ where $A = \{ a \}$ and $\mathcal{X} = \{ \mathsf{x}_1, \mathsf{x}_2, \mathsf{x}_3 \}$. Using the approach discussed above, we find the solution $h(\mathsf{x}_1) = aaaa\ aaaa$, $h(\mathsf{x}_2) = aaaa$ and $h(\mathsf{x}_3) = aa$ using the bounds $b_{\mathsf{x}_1} = 8$ and $b_{\mathsf{x}_2} = b_{\mathsf{x}_3} = 6$. We set up an automaton with $32 \cdot 38 = 1216$ states to solve the equation. In Figure 4.2 we show the computation of the SAT solver. Light grey markers indicate states considered in a run of the automaton. In this case only 261 states are needed. The dark grey markers visualize the actual path in the automaton leading to the substitution. Non-diagonal stretches are λ transitions.

4.1.3 Refining Bounds and Guiding the Search

Initial experiments revealed a major inefficiency of our approach: most of the locations were not used during the search but increased the encoding time. The many white markers in Figure 4.2, indicating unused locations, visualize this problem. Since we create all required variables $\mathsf{x} \in \mathcal{X}$ and constraints for every position $i < b_{\mathsf{x}}$, we can reduce the automaton size by lowering these upper bounds. Abstracting a word equation by the length of the variables gives us a way to refine the bounds b_{x} for some of the variables $\mathsf{x} \in \mathcal{X}$. By only considering length we obtain a Diophantine equation (see Definition 2.7) in the following manner: we assume an ordering on the variable alphabet $\mathcal{X} = \{ \mathsf{x}_0, \dots, \mathsf{x}_{n-1} \}$ and associate to each word equation variable x_j an integer variable $\bar{\mathsf{x}}_j$.

Definition 4.11. For a word equation $\alpha \doteq \beta$ with $\mathcal{X} = \{ \mathsf{x}_0, \dots, \mathsf{x}_{n-1} \}$ we define its *length abstraction* by

$$\sum_{j \in [n]} \left(|\alpha|_{\mathsf{x}_j} - |\beta|_{\mathsf{x}_j} \right) \cdot \bar{\mathsf{x}}_j = \sum_{a \in \Sigma} |\beta|_a - |\alpha|_a.$$

If a word equation has a solution $h \in \mathcal{H}_A$ then so does its length abstraction with variable $\bar{\mathsf{x}}_j = |S(\mathsf{x}_j)|$. Our interest is computing upper bounds for each variable $\mathsf{x}_k \in \mathcal{X}$ relative to the upper bounds of the bounded word equation problem. To this end consider the following natural deductions

$$\begin{aligned} &\sum_{j \in [n]} \left(|\alpha|_{\mathsf{x}_j} - |\beta|_{\mathsf{x}_j} \right) \cdot \bar{\mathsf{x}}_j = \sum_{a \in \Sigma} \left(|\beta|_a - |\alpha|_a \right) \\ \iff \bar{\mathsf{x}}_k &= \frac{\sum_{a \in \Sigma} \left(|\beta|_a - |\alpha|_a \right)}{|\alpha|_{\mathsf{x}_k} - |\beta|_{\mathsf{x}_k}} - \frac{\sum_{j \in [n] \setminus k} \left(|\alpha|_{\mathsf{x}_j} - |\beta|_{\mathsf{x}_j} \right) \cdot \bar{\mathsf{x}}_j}{|\alpha|_{\mathsf{x}_k} - |\beta|_{\mathsf{x}_k}} \\ \implies \bar{\mathsf{x}}_k &\leqslant \frac{\sum_{a \in \Sigma} \left(|\beta|_a - |\alpha|_a \right)}{|\alpha|_{\mathsf{x}_k} - |\beta|_{\mathsf{x}_k}} - \frac{\sum_{j \in \kappa} \left(|\alpha|_{\mathsf{x}_j} - |\beta|_{\mathsf{x}_j} \right) \cdot b_{\mathsf{x}_j}}{|\alpha|_{\mathsf{x}_k} - |\beta|_{\mathsf{x}_k}} = b^{\mathsf{S}}_{\mathsf{x}_k}, \end{aligned}$$

where $\kappa = \{ m \in [n] \setminus k \mid (\, |\alpha|_{\mathsf{x}_k} - |\beta|_{\mathsf{x}_k}) \cdot (|\alpha|_{\mathsf{x}_m} - |\beta|_{\mathsf{x}_m}) < 0 \}$. Whenever we have $0 < b^{\mathsf{S}}_{\mathsf{X}_k} < b_{\mathsf{X}_k}$, we use $b^{\mathsf{S}}_{\mathsf{x}_k}$ instead of b_{x_k} to prune the search space.

The length abstraction is also useful because it might give information about the unsatisfiability of an equation: if there is no solution to the

Diophantine equation, there is no solution to the word equation. We use this acquired knowledge and directly report this fact to the solver. Unfortunately whenever $|\alpha|_{\mathsf{x}} - |\beta|_{\mathsf{x}} = 0$ holds for a variable $\mathsf{x} \in \mathcal{X}$ we cannot refine the bounds, as they are not influenced by the above Diophantine equation.

The length abstraction used to refine upper bounds can also be used to guide the search in the automaton. In particular, it can restrict the allowed length of one variable based on the length of others. We refer to the coefficient of variable $\bar{\mathsf{x}}_j$ in Definition 4.11 by $\mathsf{Co}_{\alpha,\beta}(\bar{\mathsf{x}}_j) = \left(|\alpha|_{\mathsf{x}_j} - |\beta|_{\mathsf{x}_j}\right)$.

To benefit from the abstraction of the word equation inside our propositional logic encoding we use Reduced Ordered Multi-Decision Diagrams (MDD) [6]. An MDD is a directed acyclic graph, with two nodes having no outgoing edges (called `true` and `false` terminal nodes). A node in the MDD is associated to exactly one variable $\bar{\mathsf{x}}_j$, and has an outgoing edge for each element of $\bar{\mathsf{x}}_j$'s domain. In the MDD, a node labelled $\bar{\mathsf{x}}_j$ is only connected to nodes labelled $\bar{\mathsf{x}}_{j+1}$. Formally an MDD is defined as follows.

Definition 4.12. Let $\bar{\mathcal{X}} = \{\, \bar{\mathsf{x}}_0, \dots, \bar{\mathsf{x}}_{n-1} \,\}$ for $n \in \mathbb{N}$ be a set of integer variables and $D : \mathcal{X} \to \{\, d \mid d \ddot{\subset} \mathbb{Z} \,\}$ a function assigning an integer domain to each variable. A *Reduced Ordered Multi-Decision Diagrams* (MDD) is a acyclic graph $M = (V, E)$ having nodes

$$V = \{\, \mathtt{true}, \mathtt{false} \,\} \cup \{\, (\bar{\mathsf{x}}, i) \mid \bar{\mathsf{x}} \in \mathcal{X}, i \in D(\bar{\mathsf{x}}) \,\}$$

and edges

$$\begin{aligned} E \subseteq \{\, &((\bar{\mathsf{x}}_k, i), (\bar{\mathsf{x}}_{k+1}, j)) \mid k \in [n-1], \bar{\mathsf{x}}_k, \bar{\mathsf{x}}_{k+1}, \in \mathcal{X}, i \in D(\bar{\mathsf{x}}_k), j \in D(\bar{\mathsf{x}}_{k+1}) \,\} \\ &\cup \{\, ((\bar{\mathsf{x}}_{n-1}, i), \mathtt{true}), ((\bar{\mathsf{x}}_{n-1}, i), \mathtt{false}) \mid i \in D(\bar{\mathsf{x}}_{n-1}) \,\} \end{aligned}$$

such that $|\{\, ((\bar{\mathsf{x}}, i), v) \in E \mid i \in D(\bar{\mathsf{x}}) \,\}| = |D(\bar{\mathsf{x}})|$ for all $\bar{\mathsf{x}} \in \bar{\mathcal{X}}$.

For an MDD (V, E) let a *row* be a subset of nodes corresponding to a certain variable $\bar{\mathsf{x}}$, written $\mathsf{r}(\bar{\mathsf{x}}) \subseteq \{\, ((\bar{\mathsf{x}}, i), v) \in E \mid i \in D(\bar{\mathsf{x}}) \,\}$.

We create the MDD following Definition 4.11 for a word equation $\alpha \doteq \beta$ by recursively creating the rows. An MDD node is a tuple consisting of a variable $\bar{\mathsf{x}}_j$ and an integer corresponding to the partial sum which can be obtained using the coefficients and position information of all previous variables $\bar{\mathsf{x}}_k$ for $0 \leqslant k < j \in [|\mathcal{X}|]$. We introduce a new variable $\bar{\mathsf{x}}_{-1}$ labelling the initial node of the MDD. The computation is done as follows:

$$\mathsf{r}(\bar{\mathsf{x}}_i) = \left\{\, (\bar{\mathsf{x}}_i, s + k \cdot \mathsf{Co}_{\alpha,\beta}(\bar{\mathsf{x}}_i)) \mid s \in \left\{\, s' \;\middle|\; (\bar{\mathsf{x}}_{i-1}, s') \in \mathsf{r}(\bar{\mathsf{x}}_{i-1}) \,\right\}, \\ k \in [b_{\mathsf{x}_i}] \,\right\}$$

and $\mathsf{r}(\bar{\mathsf{x}}_{-1}) = \{\, (\bar{\mathsf{x}}_{-1}, 0) \,\}$. Since $\bar{\mathsf{x}}_j$ is associated to the word equations variable x_j, we let $\mathsf{r}(\mathsf{x}_j) = \mathsf{r}(\bar{\mathsf{x}}_j)$. We denote the whole set of nodes in the MDD by $M^C = \bigcup_{\mathsf{x}\in\mathcal{X}\cup\{\bar{\mathsf{x}}_{-1}\}} \mathsf{r}(\mathsf{x})$. The `true` node of the MDD is $(\bar{\mathsf{x}}_{n-1}, \mathsf{s}_\#)$, where $\mathsf{s}_\# = \sum_{a\in\Sigma} |\beta|_a - |\alpha|_a$. If the initial creation of nodes did not add this node, the given equation (Definition 4.11) is not satisfiable hence the word equation has no solution given the set bounds. Furthermore there is no need to encode the full MDD, when only a subset of its nodes can reach $(\bar{\mathsf{x}}_{n-1}, \mathsf{s}_\#)$. For reducing the MDD nodes to this subset, we calculate all predecessors of a given node $(\bar{\mathsf{x}}_i, s) \in M^C$ as follows

$$\mathsf{pred}((\bar{\mathsf{x}}_i, s)) = \left\{\, (\bar{\mathsf{x}}_{i-1}, s') \;\middle|\; s' = s - k \cdot \mathsf{Co}_{\alpha,\beta}(\bar{\mathsf{x}}_{i-1}), k \in [b_{\mathsf{x}_{i-1}}] \,\right\}.$$

The minimized set $M = F(T)$ of reachable nodes starting at the only accepting node $T = \{\, (\bar{\mathsf{x}}_{n-1}, \mathsf{s}_\#) \,\}$ is afterwards defined through a fixed point by

$$T \subseteq F(T) \wedge \Big(\forall\, p \in F(T) : q \in \mathsf{pred}(p) \wedge q \in M^C \Rightarrow q \in F(T)\Big) \qquad (*)$$

We continue by encoding this into a propositional logic formula. For that we need information on the actual length of a possible substitution. We reuse the propositional logic variables of our filled variables $\mathsf{x} \in \check{\mathcal{X}}$. The idea is to introduce $b_{\mathsf{x}} + 2$ many Boolean variables

$$\mathsf{OH}_i(0), \ldots, \mathsf{OH}_i(b_X + 1)$$

for each $\mathsf{x}_i \in \mathcal{X}$, where $\mathsf{OH}_i(j)$ is true if and only if $h(\mathsf{x}_i)$ has length j in the actual substitution $h \in \mathcal{H}_A$. This kind of encoding is known as a *one-hot encoding*. To achieve this we add a constraint forcing substitutions to have all λ in the end. This directly corresponds to the definition of the filled substitution $\check{h}$. We force our solver to adapt to this by adding clauses

$$\mathsf{K}^{\lambda}_{\mathsf{x}^{(j)}} \to \mathsf{K}^{\lambda}_{\mathsf{x}^{(j+1)}}$$

for all $j \in [b_{\mathsf{x}} - 1]$ and $\mathsf{x}^{(j)} \in \check{\mathcal{X}}$. The actual encoding is done by adding the constraints $\mathsf{OH}_i(0) \leftrightarrow \mathsf{K}^{\lambda}_{\mathsf{x}_i^{(0)}}$ and $\mathsf{OH}_i(b_{\mathsf{x}_i}) \leftrightarrow \neg\mathsf{K}^{\lambda}_{\mathsf{x}^{(b_{\mathsf{x}_i}-1)}}$,

which fix the edge cases for the substitution by the empty word and no λ inside it. For all $j \in [b_{\mathsf{x}_i}]_0$, we add the constraint

$$\mathsf{OH}_i(j) \leftrightarrow \mathsf{K}^{\lambda}_{\mathsf{X}_i^{(j)}} \wedge \neg\mathsf{K}^{\lambda}_{\mathsf{X}_i^{(j-1)}},$$

which marks the first occurrence of λ.

The encoding of the MDD is done nodewise by associating a propositional logic variable

$$\mathsf{M}_{i,j}$$

for each $i \in [|\mathcal{X}|]$, where $(\bar{\mathsf{x}}_i, j) \in M$. Our goal is now to find a path inside the MDD from node $(\bar{\mathsf{x}}_{-1}, 0)$ to $(\bar{\mathsf{x}}_{n-1}, \mathsf{s}_\#)$. Therefore we enforce a true assignment for the corresponding variables $\mathsf{M}_{-1,0}$ and $\mathsf{M}_{n-1,\mathsf{s}_\#}$ by adding

$$\left(\mathsf{M}_{-1,0} \wedge \mathsf{M}_{n-1,\mathsf{s}_\#}\right) \leftrightarrow \top.$$

A valid path is encoded by the constraint

$$\mathsf{M}_{i-1,j} \wedge \mathsf{OH}_i(k) \rightarrow \mathsf{M}_{i,s}$$

for each variable $\mathsf{x}_i \in \mathcal{X}$, $k \in [b_{\mathsf{x}_i}]_0$, where $s = j + k \cdot \mathsf{Co}_{\alpha,\beta}(\mathsf{x}_i)$ and $(\bar{\mathsf{x}}_i, s) \in M$. This encodes the fact that, whenever we are at a node $(\bar{\mathsf{x}}_{i-1}, s) \in M$ and the substitution for a variable $\bar{\mathsf{x}}_i$ has length k ($|h(\bar{\mathsf{x}}_i)| = k$), we move on to the next node, which corresponds to x_i and an integer obtained by taking the coefficient of the variable $\bar{\mathsf{x}}_i$, multiplying it by the substitution length, and adding it to the previous partial sum s.

Whenever there is only one successor to a node $(\bar{\mathsf{x}}_i, j)$ within our MDD, we directly force its corresponding one hot encoding to be true by adding

$$\mathsf{M}_{i-1,j} \rightarrow \mathsf{OH}_i(j).$$

This reduces the amount of guesses on variables.

Example 4.13. Consider the equation $\alpha \doteq \beta$ for $\alpha = a\mathsf{x}_2\mathsf{x}_0 b, \beta = a\mathsf{x}_0 a\mathsf{x}_1 \in \mathtt{Pat}_A$, where $A = \{ a, b \}$ and $\mathcal{X} = \{ \mathsf{x}_0, \mathsf{x}_1, \mathsf{x}_2 \}$. The corresponding linear equation therefore has the form $0 \cdot \bar{\mathsf{x}}_0 + (-1) \cdot \bar{\mathsf{x}}_1 + 1 \cdot \bar{\mathsf{x}}_2 = 0$ which gives us the coefficients $\mathsf{Co}_{\alpha,\beta}(\mathsf{x}_0) = 0$, $\mathsf{Co}_{\alpha,\beta}(\mathsf{x}_1) = -1$ and $\mathsf{Co}_{\alpha,\beta}(\mathsf{x}_2) = 1$. For given bounds $b_{\mathsf{x}_0} = b_{\mathsf{x}_1} = b_{\mathsf{x}_2} = 2$ the induced MDD has the form shown

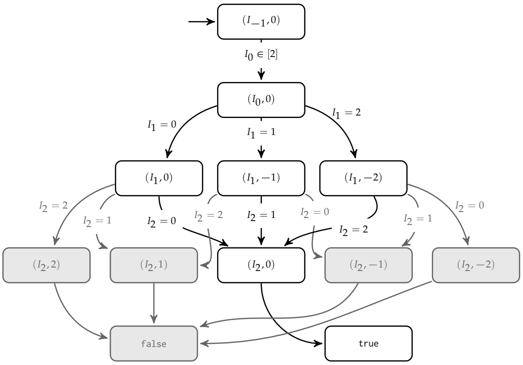


Figure 4.3. The MDD for $a\mathsf{x}_2\mathsf{x}_0b \doteq a\mathsf{x}_0a\mathsf{x}_1$

in Figure 4.3. In this example $s_\# = 0$, and therefore $(\bar{\mathsf{x}}_2, 0)$ is the only node connected to the `true` node. The minimization of the MDD by using the fixed point previously described removes all grey nodes, since they are not reachable starting at the `true` node. The solver returns the substitution $h(\mathsf{x}_0) = \varepsilon$, $h(\mathsf{x}_0) = b$ and $h(\mathsf{x}_0) = a$. It took the centred path consisting of the nodes $(\bar{\mathsf{x}}_{-1}, 0), (\bar{\mathsf{x}}_0, 0), (\bar{\mathsf{x}}_1, -1), (\bar{\mathsf{x}}_2, 0),$ `true` inside the MDD.

This optimisation does not always lead to a reduction of the state space. Whenever the occurrences of variables on both sides of the equations are the same, we cannot draw a conclusion on the optimised bounds. Nevertheless, the MDD still guides the search by reducing unnecessary guesses.

We will now show how to easily add the MDD encoding to cope with linear length constraints.

4.1.4 Adding Linear Length Constraints

Until now we have only concerned ourselves with general bounded word equations. As mentioned, bounded equations with linear constraints are of interest as well. In particular, without loss of generality we restrict to

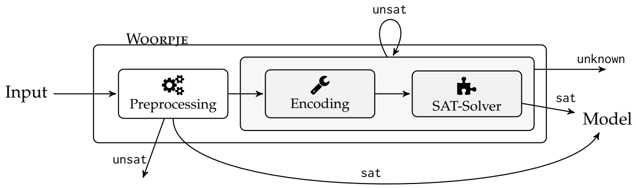


Figure 4.4. Architecture of Woorpje

linear constraints of the form (see Section 2.2.2 for details)

$$\sum_{\bar{\mathsf{x}} \in \bar{\mathcal{X}}} c_{\bar{\mathsf{x}}} \cdot \bar{\mathsf{x}} \leqslant s_{\#}$$

where $s_{\#}, c_{\bar{\mathsf{x}}} \in \mathbb{Z}$ are integer coefficents and $\bar{\mathsf{x}}_i$ are integer variables with a domain $D_i = \{\, m \in \mathbb{N} \mid 0 \leqslant m \leqslant d_i \,\}$ and a corresponding $d_i \in \mathbb{N}$. Since we are now in a similar setting to the one discussed in Section 2.3.1, each $\bar{\mathsf{x}}_i$ corresponds to the length of a substitution to a variable of the given word equation.

Notice that the structure of the linear length constraints is similar to that of Definition 4.11. For handling linear constraints we can adapt the generation of MDD nodes to keep track of the partial sum of the linear constraint, and define the accepting node of the MDD as the one where all rows have been visited and the inequality is true. We simply extend the set T which was used in the fix point iteration in (*) in the previous section to the set

$$T = \left\{ (\bar{\mathsf{x}}_{n-1}, s) \;\middle|\; (\bar{\mathsf{x}}_{n-1}, s) \in M^C \wedge s \leqslant s_{\#} \right\}.$$

After this tweak the generation of the MDD is done exactly in the same way as we previously did for the length abstraction.

4.1.5 Experimental Implementation

The approach described in the previous sections has been implemented in the tool Woorpje. The inner workings of Woorpje are visualised in Figure 4.4. Woorpje starts with a preprocessing step where obviously satisfiable, respectively, unsatisfiable word equations are immediately reported. Firstly, reduce prefixes and suffixes according to rule 1 of Lemma 3.5 (Levi's lemma). For a word equation $\alpha \doteq \beta$, if there exists pattern $\gamma, \alpha', \beta' \in \mathtt{Pat}_A$ such that $\alpha = \gamma\alpha'$ and $\beta = \gamma\beta'$. We have $\models \alpha \doteq \beta$ iff $\models \alpha' \doteq \beta'$. Therefore, it is sufficient to continue solving the latter equation. Similarly if $\alpha = \alpha'\gamma$ and $\beta = \beta'\gamma$ we can remove the suffix of the word equation $\alpha \doteq \beta$. In that sense an equation is trivially satisfiable whenever both sides are the same, meaning in the above cases $\alpha' = \beta' = \varepsilon$.

Next we use several simple lemmas from the theory of combinatorics on words to quickly determine unsatisfiability of a word equation.

Lemma 4.14. *Let $\alpha \doteq \beta$ and $\alpha, \beta \in \mathtt{Pat}_A$ be a word equation. If $\alpha \doteq \beta$ satisfies any of the following constraints, we have $\not\models \alpha \doteq \beta$.*

1. *If there exist $\gamma, \delta \in A^*$ such that $\gamma \neq \delta, |\gamma| = |\delta|$, and $(\gamma \in \text{prefix}(\alpha) \wedge \delta \in \text{prefix}(\beta))$ hold,*

2. *if $\alpha \in A^*$ and there exists $\gamma \in A$ such that $\gamma \in \text{factor}(\beta)$ and $\neg\gamma \in \text{factor}(\alpha)$ hold,*

3. *if there exists $\ell \in [\min(|\alpha|, |\beta|)]$ such that for all $1 \leqslant m \leqslant \ell$ and $\mathsf{x} \in \mathcal{X}$ we have $\mathsf{parikh}(\alpha[: m], \mathsf{x}) = \mathsf{parikh}(\beta[: m], \mathsf{x})$ but $\mathsf{parikh}(\alpha_\ell, a) \neq \mathsf{parikh}(\beta_\ell, a)$ for some $a \in A$.*

Since these lemmas are obtained from the classical literature on combinatorics on words (cf. [32]) we will omit the proofs here. Nevertheless, we will state some examples to express their usability within our implementation.

Example 4.15. The following three examples characterise the impact of Lemma 4.14.

1. Consider the word equation $ab\mathsf{x}_1 \doteq aab\mathsf{x}_2$. There is a mismatch of the second letters within the sides of the equation. Therefore, we have $\not\models ab\mathsf{x}_1 \doteq aab\mathsf{x}_2$, since we cannot find a suitable substitution.

2. Consider the word equation $ababab \doteq \mathsf{x}_1 aab\mathsf{x}_2$. We have $aab \notin \text{factor}(ababab)$ but $aab \in \text{factor}(\mathsf{x}_1 aab\mathsf{x}_2)$.

3. Consider the word equation $a\mathsf{x}_1 \doteq \mathsf{x}_1 b$. We have a mismatch within the Parikh vector at index 2 with respect to the letters a and b. Visualising

the Parikh vector for all suitable length has the following form:

$$\begin{array}{c|cc} & 1 & 2 \\ \hline a & 1 & 1 \\ b & 0 & 0 \\ x_1 & 0 & 1 \end{array} \qquad \text{and} \qquad \begin{array}{c|cc} & 1 & 2 \\ \hline a & 0 & 0 \\ b & 0 & 1 \\ x_1 & 1 & 1 \end{array}$$

After the preprocessing step, Woorpje starts an iterative search based on the length of a possible solution for a variable of the given constraints. If a solution is found, it is reported. The maximal value of i is user definable, and by default set to 2^n where n is the length of the given equation. If Woorpje reaches the given bound without a verdict, it returns unknown. Currently the search is starting at $1^2 = 1$. The idea of just going up, in our search, to a single-exponential bound on the length of the solutions is supported by the widely believed conjecture that the satisfiability of word equations is in NP and the length of the solution of an equation is at most exponential in the length of the equation.

The current bound $B \in \mathbb{N}$ defines the upper bounds of the domains for our variables, meaning for each $\mathsf{x} \in \mathcal{X}$ we have $0 \leqslant |h(\mathsf{x})| \leqslant B$. Based on these bounds we use the length abstraction of Definition 4.11 to tighten the bounds. Our new bounds are now used to build the MDD running Algorithm 1 which is an implementation of the steps explained in Section 4.1.3. We build each row of the MDD going from top to bottom, while removing nodes which are not reachable from the bottom on the fly. The MDD might potentially allow us to optimise the bounds for each solution again.

Lastly, we encode the resulting information of the MDD and the word equations using the optimised bounds into a propositional logic formula and solve it with Glucose Audemard and Simon [10] for increasing maximal variable lengths ($i^2 = B$, where i is the current iteration). Note that our approach is not skipping any possible smaller substitutions because of the defined λ-padding in the images of the variables under a substitution. As soon as we find a valid solution to the equation, we report it. Whenever we find a bound-independent argument for the unsatisfiability of the equation, we print it as well.

Algorithm 1: Calculation of the MDD for a bounded word equation $\alpha \doteq \beta$

```
Data: $\alpha \doteq \beta \in \mathtt{Pat}_A$, $\mathcal{L} : \mathcal{X} \to \mathbb{N}$
Result: nodes $v$, edges $e$
$r :=$ dictionary();
$i := 0$;
$v, e := \emptyset, \emptyset$;
$r[\bar{x}_{-1}] := \{ (\bar{x}_{-1}, 0) \}$;
while $i < [|\mathcal{X}|]$ do
    $preds := \emptyset$;
    $tmp\_edges := \emptyset$;
    forall $(\bar{x}_{i-1}, s) \in r(\bar{x}_{i-1})$ do
        forall $k \in [\mathcal{L}(\bar{x}_i)]$ do
            $s_f := s + k \cdot \mathsf{Co}_{\alpha,\beta}(\bar{x}_i)$;
            $s_p := s - k \cdot \mathsf{Co}_{\alpha,\beta}(\bar{x}_i)$;
            $r[\bar{x}_i] := \{ (\bar{x}_i, s_f) \}$;
            $preds := preds \cup \{ (\bar{x}_{i-1}, s_p) \}$;
            $tmp\_edges := tmp\_edges \cup \{ ((\bar{x}_{i-1}, s), (\bar{x}_i, s_f)) \}$;
    $r[\bar{x}_{i-1}] := r[\bar{x}_{i-1}] \cap preds$;
    $e := e \cup \{ (v_1, v_2) \in tmp\_edges \mid v_1 \in r[\bar{x}_{i-1}] \}$;
    $i := i + 1$;
$j := -1$;
while $j < [|\mathcal{X}| - 1]$ do
    forall $(\bar{x}, s) \in r[(\bar{x}_j)]$ do
        $v := v \cup \bar{x}$;
```

4.1.6 Releated Work

To the best of our knowledge directly encoding string constraints into a propositional logic formula has not been evaluated before. There exist similar approaches like the tools Kaluza [99] or Hampi [71] which target solving bounded word equations. Rather than encoding string constraints directly, both tools encode the given input formula into bit vectors. No doubt, our techniques are inspired approaches coming from the area of CP-solving (cf. [9] and references therein). Especially the usage of MDDs to encode linear length constraints is a common practice within the CP world. Overall, our approach combines these areas nicely and enriches the string solving community in an unseen way.

4.1.7 Conclusion

In this section we presented a method for solving word equations by using a SAT-Solver. Our idea shows that we can draw certain benefits by carefully encoding a specific problem into propositional logic, while exploiting problem-specific insights, instead of using a more general approach.

Since we currently only implement a preprocessing technique to check for unsatisfiability, our next step is the enrichment of this part. In the future, we aim to extend our approach to include regular constraints. As our approach relies on automata theory, it is expected that this is achievable. Another step is the enrichment of our linear constraint solving. We currently do a basic analysis by using MDDs. There are a few refinement steps described in [6] which seem applicable. A next major step is to develop a more efficient encoding of the alphabet of constants. Currently the state space explodes due to the massive branching caused by the usage of large alphabets. The same problem occurs in other approaches to solving word equations (such as the recompression technique [66]), and it seems that overcoming it could require some new significant theoretical insights from which all approaches will benefit.

4.2 Rule-based Solving of String Contraints

The most successful string solvers, according to the latest results reported by the developers, are all based on the idea of integrating the word equation solver into an SMT solver which can then be naturally used by standard software verification tools. The integration into an SMT solver has influenced the solving strategy, and it seems that solving length-constrained word equations is mostly based on enhancing the set of initial linear equations by a series of other linear length constraints induced by the word equations, and then using the very efficient arithmetic solver of the SMT solver on this enlarged linear system.

In this section we describe a different path: we solve word equations enhanced with length constraints, meaning we consider the theory established in Section 2.3.1, but have solving the word equation by word-oriented transformations as our main focus and only update the existing system of linear constraints after each transformation.

As such we present a transformation-system for solving systems of word equations with length constraints. Our system is built starting from *Levi's lemma* which we discussed earlier in Section 3.5.2. We enrich the initial rules being presented, inspired by combinatorial results from the theory of word equations (see, e.g., [32]) and then extend them to work in the framework of linear length constraints. On top of this, we apply further reduction steps to keep the search space of our system as small as possible.

We also enhance our transformation-system by integrating three of the major SMT solvers handling word equations, that are CVC4 [16], Z3STR3 [23] and Z3SEQ [45], which we call according to empirically found heuristics. We implemented these ideas and show, by running our implementation on a set of benchmarks, that the aforementioned state of the art solvers are largely improved by our approach.

This section is organised as follows: we introduce our transition system including the handling of length constraints. Afterwards, we describe the design of our transitions rules for the aforementioned system. The rules are split into two categories: 1. rules influenced by Levi's lemma, and 2. simplification rules. Moreover, in the preceding section, we explain how we identify our terminating nodes within the induced graph of the transition system. Before concluding, we explain how we implemented our approach and briefly talk about related work.

4.2.1 Transformation Systems

In this section, we present the framework which forms the basis for our approach. We introduce it for systems of word equations first and, subsequently, extend it to support systems of word equations with length constraints.

As the starter we, define a partial function r which allows us to reason about certain states of our ultimate solution to a word equation within our transformation system.

Definition 4.16. Let $E, E' \mathrel{\ddot{\subset}} \mathsf{WEQ}_A$ be systems of word equations. A partial function $r : \mathcal{X} \rightharpoonup \mathtt{Pat}_A$ is called an *r-replacement* from E' to E if

1. if $r(\mathsf{x}_0)$ is defined then for all $\mathsf{x}_1 \in \mathcal{X} \setminus \{ \mathsf{x}_0 \}$ we have $|r(\mathsf{x}_1)|_{\mathsf{x}_0} = 0$, and
2. whenever there exists a $h' \in \mathcal{H}_A$ such that $h' \models E'$ then $h' \oplus r \models E$ where

$$\left(h' \oplus r\right)(\mathsf{x}_0) = \begin{cases} h'(r(\mathsf{x}_0)) & \text{if } \mathsf{x}_0 \in \mathsf{dom}(r), \\ h'(\mathsf{x}_0) & \text{otherwise.} \end{cases}$$

We denote this relationship between r, E and E' by $(E \Bumpeq_r E')$.

An r-replacement naturally extends to pattern $\mathtt{Pat}_A$. A partial function $r : \mathtt{Pat}_A \rightharpoonup \mathtt{Pat}_A$ simply has to fulfil the requirements of a morphism, meaning $r(a) = a$ for all $a \in A_\varepsilon$ and for pattern $\alpha, \beta \in \mathtt{Pat}_A$ we have $r(\alpha\beta) = r(\alpha)r(\beta)$, as well as the requirements made in the above definition. To ease understanding we denote the r-replacement for a pattern $\alpha \in \mathtt{Pat}_A$ by

$$r(\!(\alpha)\!).$$

This operation replaces all occurrences of x_0 by $r(\mathsf{x}_0)$, when it is defined. We generalise this to a system of word equations E by

$$r(\!(E)\!) = \{ r(\!(\alpha)\!) \doteq r(\!(\beta)\!) \mid \alpha \doteq \beta \in E \}.$$

To grasp this definition we present the application of an r-replacement in an example.

Example 4.17. Consider the system of word equations $E = \{ \mathsf{x}_0 b \doteq a\mathsf{x}_1 \}$, the partial function $r = \{ \mathsf{x}_0 \mapsto a\mathsf{x}_0 \}$ and the system of word equations $E' = \{ \mathsf{ax}_0 b \doteq a\mathsf{x}_1 \}$ which has a solution $h' = \{ \mathsf{x}_0 \mapsto a, \mathsf{x}_1 \mapsto ab \} \in \mathcal{H}_A$. Then condition 1 holds since $r(\mathsf{x}_1)$ is not defined. Condition 2 holds as well since

$$h'(r(\mathsf{x}_0))b = h'(a\mathsf{x}_0)b = aab = ah'(\mathsf{x}_1) = ah'(r(\mathsf{x}_1))$$

and therefore $h' \oplus r \models E$.

In this section we use $\mathtt{SAT} \subseteq \mathsf{WL}_A$ (respectively $\mathtt{UNSAT} \subseteq \mathsf{WL}_A$) to denote the set of all satisfiable (respectively unsatisfiable) word equations with (a possibly trivial set of) length constraints.

We are now ready to define the framework of this section: the transition system.

Definition 4.18 (Transformation System)**.** Let $\mathcal{X}$ be a finite set of variables, then a transformation system is a tuple $(S, \rightarrow, \mathbb{S}, \mathbb{U})$ where

1. $S = 2^{\mathsf{WEQ}_A}$ is the set of states,
2. $\rightarrow \subseteq S \times \{ \mathcal{X} \rightharpoonup \mathtt{Pat}_A \} \times S$ with $(E, r, E') \in \rightarrow$ implies that $(E \Bumpeq_r E')$ is a transformation relation,
3. $\mathbb{S} \subseteq \mathtt{SAT}$ a set of word equations guaranteed to be satisfied, and
4. $\mathbb{U} \subseteq \mathtt{UNSAT}$ a set of word equations guaranteed not to be satisfied.

To ease readability we write $E \xrightarrow{r} E'$ whenever $(E, r, E') \in \rightarrow$. If $E = \{ \alpha \doteq \beta \}$ and $E' = \{ \alpha' \doteq \beta' \}$ we write $\alpha \doteq \beta \xrightarrow{r} \alpha' \doteq \beta'$ instead of $\{ \alpha \doteq \beta \} \xrightarrow{r} \{ \alpha' \doteq \beta' \}$.

The modification of the r function seen in Example 4.17 is directly correlating to a transformation step $E \xrightarrow{r} E'$ introduced in the above definition. The ultimate goal is constructing a substitution to an initial system of word equations by performing transformation steps until we reach a trivially satisfiable system of word equations. Afterwards we construct the solution backwards by iteratively applying the $\oplus$ operation starting with the solution $h = \emptyset$. The following examples gives an intuition of our transition system and the described intentional behaviour.

Example 4.19. Let $(S, \rightarrow, \mathbb{S}, \mathbb{U})$ be a transformation system. Let the system of word equations $E = \{ \mathsf{x}_0 b \doteq a \mathsf{x}_1 \}$, $E' = \{ a \mathsf{x}_2 b \doteq a \mathsf{x}_1 \}$, $E'' = \{ aab \doteq aab \} \in S$. Consider the partial functions $r = \{ \mathsf{x}_0 \mapsto a \mathsf{x}_2 \}$ and $r' = \{ \mathsf{x}_2 \mapsto a, \mathsf{x}_1 \mapsto ab \}$ such that $E \xrightarrow{r} E'$ and $E' \xrightarrow{r'} E''$. Clearly, we have $h'' = \emptyset \models E''$. We can obtain the solution h such that $h \models E$ by using the $\oplus$ operator as follows:

$$h = (h'' \oplus r') \oplus r.$$

Therefore we get

$$h(\mathsf{x}_0) = h(r'(r(\mathsf{x}_0))) = h''(r'(a\mathsf{x}_2)) = h(aa) = aa$$

and

$$h(\mathsf{x}_1) = h(r'(r(\mathsf{x}_1))) = h''(r'(ab)) = h(ab) = ab.$$

To cope with length constraints we modify our transition-system given in Definition 4.18 as follows:

Definition 4.20 (Extended Transformation System). Let $\mathcal{X}$ be a finite set of variables, then a transformation system is a tuple $(S, \rightarrow, \mathbb{S}, \mathbb{U})$ where

1. $S = 2^{\mathsf{WEQ}_A \times \mathsf{LinC}}$ is the set of states,

2. $\rightarrow \subseteq S \times \{ \mathcal{X} \rightharpoonup \mathtt{Pat}_A \} \times S$ with $((E,L), r, (E', L')) \in \rightarrow$ implies $(E \Bumpeq_r E')$ is a transformation relation and for each $(\sum_{\mathsf{x}_i \in \mathcal{X}} c_i \bar{\mathsf{x}}_i \leqslant c) \in L$ we have

$$\sum_{\mathsf{x}_i \in \mathsf{dom}(r)} c_i \cdot \left| r(\mathsf{x}_i)_{|\{\mathsf{x}_i\}} \right| \cdot \bar{\mathsf{x}}_i \tag{1}$$

$$+ \sum_{\mathsf{x}_i \in \mathcal{X} \setminus \mathsf{dom}(r)} \left(c_i + \sum_{\mathsf{x}_j \in \mathsf{dom}(r)} \left(c_j \cdot \left| r(\mathsf{x}_j)_{|\{\mathsf{x}_i\}} \right| \right) \right) \cdot \bar{\mathsf{x}}_i \tag{2}$$

$$\leqslant c - \left(\sum_{\mathsf{x}_i \in \mathsf{dom}(r)} c_i \cdot \left| r(\mathsf{x}_i)_{|A} \right| \right) \tag{3}$$

in L',

3. $\mathbb{S} \subseteq \mathsf{SAT}$ a set of word equations guaranteed to be satisfied, and

4. $\mathbb{U} \subseteq \mathsf{UNSAT}$ a set of word equations guaranteed to not be satisfied.

We modify the set of states in the new transformation system and require that the linear constraint part is modified to accommodate for the transformations performed on word equation part. In particular, $(E, L) \xrightarrow{r} (E', L')$ is a valid transformation rule if $E \xrightarrow{r} E'$ and L' modified according to the definition. Thereby, part (1) handles the variables modified by r. Intuitively, we count the self-replacements of a variable. Whenever a variable of the domain of r occurs a least once in its image, the linear length constraint is affected accordingly. Part (2) takes care of possible occurrences of variables in the image of r, not being present within the domain. Whenever a variable $\mathsf{x} \in \mathsf{dom}(r)$ is affected by the image, the linear length constraint needs a modification according to the coefficient of the domain's variable. The right hand side of the linear constraints is simply the old right hand side c from which we subtract all terminal symbols within the image of r. These are the positions inside the given system of word equations which are fixed.

We briefly give an intuition of how this inequality is derived. Let $\mathcal{X} = \{ \mathsf{x}_1, \ldots, \mathsf{x}_n \}$, $c_1 \bar{\mathsf{x}}_1 + \ldots + c_n \bar{\mathsf{x}}_n \leqslant c \in L$ and $\mathsf{dom}(r) \subseteq \mathcal{X}$. By using

Definition 4.11 we obtain

$$\bar{r}(\mathsf{x}_i) = |r(\mathsf{x}_i)_{|A}| + \sum_{\mathsf{x}\in\mathcal{X}} |r(\mathsf{x}_i)_{|\{\,\mathsf{x}\,\}}| \cdot \bar{\mathsf{x}}$$

for $\mathsf{x}_i \in \mathsf{dom}(r)$ forms a length generalisation of the image of r. For an arbitrary $\mathsf{x}_i \in \mathsf{dom}(r)$ we have:

$$\begin{aligned} & c_1\bar{\mathsf{x}}_1 + \ldots + c_i\bar{r}(\mathsf{x}_i) + \ldots + c_n\bar{\mathsf{x}}_n \leqslant c \\ \Longleftrightarrow\; & c_1\bar{\mathsf{x}}_1 + \ldots + c_i \left(|r(\mathsf{x}_i)_{|A}| + \sum_{x\in\mathcal{X}} |r(\mathsf{x}_i)_{|\{\,x\,\}}| \cdot \bar{\mathsf{x}}_0 \right) + \ldots + c_n\bar{\mathsf{x}}_n \\ & \leqslant c \\ & \leqslant c - c_i \cdot |r(\mathsf{x}_i)_{|A}| \end{aligned}$$

Which indicates the correctness of our transformations purely done based on the r function for given linear constraints.

Example 4.21. Let $T = (S, \rightarrow, \mathbb{S}, \mathbb{U})$ be a transformation system such that $S = \mathsf{WL}_A$. Let the system of word equations $E = \{\, \mathsf{x}_0 b \doteq a\mathsf{x}_1 \,\}$,$E' = \{\, \mathsf{a}\mathsf{x}_0 b \doteq a\mathsf{x}_1 \,\}$. Consider the partial function $r = \{\, \mathsf{x}_0 \mapsto a\mathsf{x}_0 \,\}$ such that $E \xrightarrow{r} E'$ is a valid transformation and the set of linear length constraints $L = \{\, 2 \cdot \bar{\mathsf{x}}_0 + 1 \cdot \bar{\mathsf{x}}_1 \leqslant 5 \,\}$, then there exists a transformation step in (E, L) $\xrightarrow{r}$ (E', L') in T with L' the set of linear length constraints obtained by the above given rules as follows:

$$\begin{aligned} & c_0 \cdot \left| r(\mathsf{x}_0)_{|\{\mathsf{x}_0\}} \right| \cdot \bar{\mathsf{x}}_0 && (1) \\ + & \left(c_1 + c_0 \cdot \left| r(\mathsf{x}_0)_{|\{\mathsf{x}_1\}} \right| \right) \cdot \bar{\mathsf{x}}_1 && (2) \\ & \leqslant 5 - c_0 \cdot \left| r(\mathsf{x}_0)_{|A} \right| && (3) \end{aligned}$$

and substituting variables $(2 \cdot 1) \cdot \bar{\mathsf{x}}_0 + (1 + 2 \cdot 0) \cdot \bar{\mathsf{x}}_1 \leqslant 5 - 2 \cdot 1$ which leads to the actual modified length constraint

$$l' = 2 \cdot \bar{\mathsf{x}}_0 + 1 \cdot \bar{\mathsf{x}}_1 \leqslant 3$$

such that $L' = \{\, l' \,\}$.

If we are given a transformation system $(S, \rightarrow, \mathbb{S}, \mathbb{U})$, then solving a system of word equations (systems of word equations with length constraints) E_0 equates to finding a transformation sequence

$$E_0 \xrightarrow{r_0} E_1 \xrightarrow{r_1} \dots \xrightarrow{r_{n-1}} E_n,$$

where $E_n \in \top$. Whenever linear constraints are involved we solve a system of linear equations over integers to determine $E_n \in \mathbb{S}$. Traversing this sequence of transformations backwards generates a solution for E_0, by obtaining a solution for E_{i-1} from the solution for E_i, for all $1 \leqslant i \leqslant n$. The following Lemma captures this explanation and proves soundness and completeness of the transformation system.

Lemma 4.22. *Given a transformation system $T = (S, \rightarrow, \mathbb{S}, \mathbb{U})$ and a system of word equations (systems of word equations with length constraints) E_0, we say T is* sound *if*

1. *there is a sequence $s = E_0 \xrightarrow{r_0} E_1 \xrightarrow{r_1} \dots \xrightarrow{r_{n-1}} E_n$, with $E_n \in \mathbb{S}$, implies $\models E_0$, and*
2. *for all sequences $s = E_0 \xrightarrow{r_0} E_1 \xrightarrow{r_1} \dots \xrightarrow{r_{n-1}} E_n$, with $E_n \in \mathbb{U}$, implies $\not\models E_0$,*

Proof. A sequence of length 0 implies $E_0 \in \mathbb{S}$. By definition there exists a solution h such that $h \models E_0$. Therefore, $\models E_0$. Now let $s = E_0 \xrightarrow{r_0} E_1 \xrightarrow{r_1} \dots \xrightarrow{r_n} E_{n+1}$ such that $E_{n+1} \in \mathbb{S}$. By definition there exists a solution h such that $h \models E_{n+1}$. Since s is a valid sequence in T we obtain a solution $h' = h \oplus r_n$ such that $h' \models E_n$. By induction hypothesis we get $\models E_0$.

Since $\models E_0$ there exists a solution h such that $h \models E_0$. Consider the system of word equations $E = h((E_0))$. Since h maps the solution from E_0 to E there is a transformation step $E_0 \xrightarrow{h} E$ in T. Moreover, we have $\alpha, \beta \in A^*$ for all $\alpha \doteq \beta \in E$. Therefore, $E \in \mathbb{S}$.

The proof for part two follows analogously. □

The setup of our transformation system essentially states the question whether we can find a path within the graph leading to a trivial system of word equations (possibly including linear length constraints). Thus, we are targeting a classical reachability problem. In that sense the core of our technique is relying of a reachability algorithm given in Algorithm 2. Notice that applying Algorithm 2 for a *sound* transformation does guarantee correct results being returned, but it does not guarantee termination.

Algorithm 2: Classic reachability algorithm. Equations that have not been explored (but found) are kept in the set `Waiting`, and states that have already been processed are kept in `Passed`.

```
Data: Equation System: $E$
Passed := $\emptyset$;
Waiting := $\{E\}$;
while Waiting $\neq \emptyset$ do
    Let $E_c \in$ Waiting;
    Waiting := Waiting$\setminus\{E_c\}$;
    Passed := Passed $\cup$ $\{ E_c \}$;
    if $E_c \in \mathbb{S}$ then
        return tt
    if $E_c \notin \mathbb{U}$ then
        Waiting := Waiting $\cup$ $\{E' \mid E_c \xrightarrow{r} E'\}$;
        Waiting := Waiting$\setminus$Passed
return ff
```

In the case of systems of pure word equations (without any other type of constraints) termination can also be shown for an instantiation of the system. However, in the case of equations with length constraints, showing the termination of our algorithm, for a particular transformation system, would solve the aforementioned open problem of deciding word equations with length constraints.

In order to solve a system of word equations we need to define a transformation system as discussed in this section. To target the actual goal we have to define three parts, namely

1. transformation rules,
2. identifying the set of satisfiable system of word equations $\mathbb{S}$, and
3. identifying the set of unsatisfiable system of word equations $\mathbb{U}$.

In the next section we start by defining the transformation rules of our transformation system.

4.2.2 Transformation rules

The transformation rules ($\rightarrow$) used in our setup are split into two categories: a set ($\rightarrow_{\mathrm{L}}$) developed around *Levi's lemma* and a set ($\rightarrow_{\mathrm{S}}$) based on simple rules about prefixes and suffixes of words. We describe both of them in the following, but note that they are unified into $\to = \to_{\mathrm{L}} \cup \to_{\mathrm{S}}$. In both cases, we provide the transformation rules for general word equations but, as previously seen in Definition 4.20, they can be adapted to word equations with length constraints.

The rules $\to_{\mathrm{L}}$

Given a word equation $\alpha \doteq \beta$, Levi's lemma makes an assumption regarding the prefix (suffix) of the possible solution $h \in \mathcal{H}_A$: either $h(\alpha)$ is a prefix (suffix) of $h(\beta)$ or vice versa. In Section 3.5.2 we discussed all details with respect to this famous lemma. The following definition is an adaptation of the mentioned rules to our transformation system.

Definition 4.23. Let E be a system of word equations and let $\mathtt{x}_2 \in \mathcal{X}$ be a fresh variable not occurring in E.

For $E = E' \cup \{\mathtt{x}_0\alpha \doteq \mathtt{x}_1\beta\}$ we have the following rules.

$$E \xrightarrow{r}_{\mathtt{l}_1^{\mathtt{pre}}} r((E')) \cup \{\, \mathtt{x}_1\mathtt{x}_2 r((\alpha)) \doteq \mathtt{x}_1 r((\beta)) \,\} \qquad (r = \{\, \mathtt{x}_0 \mapsto \mathtt{x}_1\mathtt{x}_2 \,\})$$

$$E \xrightarrow{r}_{\mathtt{l}_2^{\mathtt{pre}}} r((E')) \cup \{\, \mathtt{x}_0 r((\alpha)) \doteq \mathtt{x}_0\mathtt{x}_2 r((\beta)) \,\} \qquad (r = \{\, \mathtt{x}_1 \mapsto \mathtt{x}_0\mathtt{x}_2 \,\})$$

$$E \xrightarrow{r}_{\mathtt{l}_3^{\mathtt{pre}}} r((E')) \cup \{\, \mathtt{x}_1 r((\alpha)) \doteq \mathtt{x}_1 r((\beta)) \,\} \qquad (r = \{\, \mathtt{x}_0 \mapsto \mathtt{x}_1 \,\})$$

For $E = E' \cup \{\alpha\mathtt{x}_0 \doteq \beta\mathtt{x}_1\}$ we have the following rules:

$$E \xrightarrow{r}_{\mathtt{l}_1^{\mathtt{suf}}} r((E')) \cup \{\, r((\alpha))\mathtt{x}_2\mathtt{x}_1 \doteq r((\beta))\mathtt{x}_1 \,\} \qquad (r = \{\, \mathtt{x}_0 \mapsto \mathtt{x}_2\mathtt{x}_1 \,\})$$

$$E \xrightarrow{r}_{\mathtt{l}_2^{\mathtt{suf}}} r((E')) \cup \{\, r((\alpha))\mathtt{x}_0 \doteq r((\beta))\mathtt{x}_2\mathtt{x}_0 \,\} \qquad (r = \{\, \mathtt{x}_1 \mapsto \mathtt{x}_2\mathtt{x}_0 \,\})$$

$$E \xrightarrow{r}_{\mathtt{l}_3^{\mathtt{suf}}} r((E')) \cup \{\, r((\alpha))\mathtt{x}_1 \doteq r((\beta))\mathtt{x}_1 \,\} \qquad (r = \{\, \mathtt{x}_0 \mapsto \mathtt{x}_1 \,\})$$

For $E = E' \cup \{\mathtt{x}_0\alpha \doteq a\beta\}$, with $a \in A$, the rules are:

$$E \xrightarrow{r}_{\mathtt{l}_4^{\mathtt{pre}}} r((E')) \cup \{\, a\mathtt{x}_2 r((\alpha)) \doteq a r((\beta)) \,\} \qquad (r = \{\, \mathtt{x}_0 \mapsto a\mathtt{x}_2 \,\})$$

$$E \xrightarrow{r}_{\mathtt{l}_5^{\mathtt{pre}}} r((E')) \cup \{\, r((\alpha)) \doteq a r((\beta)) \,\} \qquad (r = \{\, \mathtt{x}_0 \mapsto \varepsilon \,\})$$

For $E = E' \cup \{\alpha\mathtt{x}_0 \doteq \beta a\}$, with $a \in A$, the rules are as follows:

$$E \xrightarrow{r}_{\mathtt{l}_4^{\mathtt{suf}}} r((E')) \cup \{\, r((\alpha))\mathtt{x}_2 a \doteq r((\beta))a \,\} \qquad (r = \{\, \mathtt{x}_0 \mapsto \mathtt{x}_2 a \,\})$$

$$E \xrightarrow{r}_{\mathtt{l}_5^{\mathtt{suf}}} r((E')) \cup \{\, r((\alpha)) \doteq r((\beta))a \,\} \qquad (r = \{\, \mathtt{x}_0 \mapsto \varepsilon \,\})$$

We define $\rightarrow_{\mathtt{L}} = \cup_{x \in \{\, \mathtt{l_i}^{\mathtt{pre}}, \mathtt{l_i}^{\mathtt{suf}} \mid i \in [6]_0 \,\}} \rightarrow_x$.

Nearly each rule introduces a new variable $\mathsf{x}_2 \in \mathcal{X}$, but removes a present variable (namely x_0 from the ones used above). The number of variables never increases, which, in practice, allows reusing eliminated variables without introducing new ones. The following examples gives an intuition on our transformation rules and clarifies how we reuse variables.

Example 4.24. Consider the system of word equations $E = \{\, \mathsf{x}_0 ab \mathsf{x}_1 \doteq a \mathsf{x}_0 \mathsf{x}_1 b, \mathsf{x}_0 b \doteq a \mathsf{x}_1 \,\}$. The equation $\mathsf{x}_0 ab \mathsf{x}_1 \doteq a \mathsf{x}_0 \mathsf{x}_1 b$ allows applying rule 14, which gives the mapping $r = \{\, \mathsf{x}_0 \rightarrow a \mathsf{x}_0 \,\}$. The reachable state is

$$\begin{aligned} E &= \{\, a \mathsf{x}_0 r((ab \mathsf{x}_1)) \doteq a r((\mathsf{x}_0 \mathsf{x}_1 b)), r((\mathsf{x}_0 b)) \doteq r((a \mathsf{x}_1)) \,\} \\ &= \{\, a \mathsf{x}_0 ab \mathsf{x}_1 \doteq aa \mathsf{x}_0 \mathsf{x}_1 b, a \mathsf{x}_0 b \doteq a \mathsf{x}_1 \}. \end{aligned}$$

We reused x_0 instead of introducing a new variable x_2.

As observed in Example 3.6 the graph induced by the transition rules might become huge, infeasible for our computers. Therefore, we introduce assisting rules to stop the search early. We do so by observing syntactical patterns within our systems of word equations where either a solution can be directly obtained or none exists.

The rules $\rightarrow_{\mathtt{S}}$

These rules are used for simplifying system of word equations. We state them for prefixes only (indicated by `pre`), but they apply analogously to suffixes (indicated by `suf`).

Lemma 4.25. *Given a system of word equations E, for $e = \alpha_1 \alpha_2 \doteq \beta_1 \beta_2 \in E$ we have:*

1. $E \xrightarrow{\varnothing}_{\mathtt{s}_1} E \setminus \{\, e \,\}$, *if* $\alpha_1 \alpha_2 = \beta_1 \beta_2$,
2. $E \xrightarrow{\varnothing}_{\mathtt{s}_2^{\mathtt{pre}}} E \setminus \{\, e \,\} \cup \{\, \alpha_2 \doteq \beta_2 \,\}$ *if* $\alpha_1 = \beta_1$,
3. $E \xrightarrow{\varnothing}_{\mathtt{s}_3} E \setminus \{\, e \,\} \cup \{\, \alpha_1 \doteq \beta_1, \alpha_2 \doteq \beta_2 \,\}$ *if* $|\alpha_1|_{\mathsf{x}_0} = |\beta_1|_{\mathsf{x}_0}$ *for all* $\mathsf{x}_0 \in \mathcal{X}$ *and* $(\sum_{a \in A} |\alpha_1|_a - |\beta_1|_a) = 0$ *, and*
4. $E \xrightarrow{r}_{\mathtt{s}_4^{\mathtt{pre}}} r((E \setminus \{\, e \,\}))$ *where* $r = \{\, \alpha_1 \mapsto \beta_1 \beta_2 \,\}$ *if* $\alpha_1 \in \mathcal{X}$ *and* $\alpha_2 = \varepsilon$.

The proofs of these simplification rules are omitted since they directly follow by simple length arguments. In fact, we gave an intuition on the first two rules in the beginning of Section 4.1.5 since we used similar arguments in our SAT solving based approach (whenever applicable) in the previous section.

In practice, these rules are never applied individually but always used in a sequence until no more rules can be used. The iteration of these rules until a fix-point is reached is defined as follows.

Definition 4.26. Let $\rightsquigarrow_{\mathtt{S}} = \bigcup_{x \in \left\{ \mathtt{S}_1, \mathtt{S}_2^{\mathtt{pre}}, \mathtt{S}_2^{\mathtt{suf}}, \mathtt{S}_3, \mathtt{S}_4^{\mathtt{pre}}, \mathtt{S}_4^{\mathtt{suf}} \right\}} \rightarrow_x$ be the joint set of simplification rules and

$$\mathsf{simp} = \left\{ E \,\ddot{\subset}\, \mathsf{WEQ}_A \;\middle|\; \neg\exists E', r : E \overset{r}{\rightsquigarrow}_{\mathtt{S}} E' \right\}$$

be the set of system of word equations which can not be simplified further with respect to the given rules. The complete simplification rule $(\rightarrow_{\mathtt{S}})$ from a system of word equations E_0 to E_k for $k \in \mathbb{N}$ is defined by $E_0 \xrightarrow{r}_{\mathtt{S}} E_k$ if $E_k \in \mathsf{simp}$, and there exists a trace of simplifications $E_i \overset{r_{i+1}}{\rightsquigarrow}_{\mathtt{S}} E_{i+1}$ for $i \in [k]$ and $r = r_k \circ r_{k-1} \circ \ldots \circ r_1$.

As previously mentioned, our approach can be seen as a reachability problem in a graph whose nodes are the systems of equations we can obtain via the described transformations. Empirical observations suggest that, at least for regular word equations with less than 10 variables (i.e., equations where each variable occurs at most twice, once in every side), the diameter of the graphs is low, and the graphs are typically very well connected. So, in a sense, the path from the initial node to the node that enables us to make a decision should be short. For certain regular equations, it can also be observed that applying Levi's lemma to both prefixes and suffixes of the equation has the effect of increasing the branching factor of vertices, and the number of vertices, without necessarily having any positive effect on the diameter, and thus it might be expected that applying the transformations only in one direction is more efficient without restricting the set of possible solutions. Interestingly enough, our empirical studies have shown that it is (from the point of view of the run-time) more efficient to apply both prefix and suffix rules.

We will now review the simplification rules within an example.

Example 4.27. Consider the system of word equations $E = \{\, \mathtt{x}_0 \doteq ab, a\mathtt{x}_0 \doteq \mathtt{x}_1 b \,\}$. By using rule $\mathtt{s}_4^{\mathtt{pre}}$ to the word equation $\mathtt{x}_0 \doteq ab$ we derive the function $r = \{\, \mathtt{x}_0 \mapsto ab \,\}$. This leads to the successor $E' = r((E \backslash \{\, \mathtt{x}_0 \,\} \doteq ab)) = r((\{\, a\mathtt{x}_0 \doteq \mathtt{x}_1 b \,\})) = \{\, aab \doteq \mathtt{x}_1 b \,\}$. Applying rule $\mathtt{s}_4^{\mathtt{suf}}$ leads to $E'' = \{\, aa \doteq \mathtt{x}_1 \,\}$. The system of word equations allows another application of $\mathtt{s}_4^{\mathtt{pre}}$ which results in a function $r' = \{\, \mathtt{x}_1 \mapsto aa \,\}$ and $E''' = \emptyset$. Backtracking the solution as seen in Example 4.19 gives us $h = \{\, \mathtt{x}_0 \mapsto ab, \mathtt{x}_1 \mapsto aa \,\}$ which is indeed a solution to E.

The simplification rules potentially decrease the length of a path to a trivially satisfiable or unsatisfiable system of word equations. This can be seen as short cuts within our search. The next step is raising the well known unsatisfiable systems of word equations within our transition system to speed up the search.

4.2.3 Identification of the sets $\mathbb{U}$ and $\mathbb{S}$

The following conditions determine the unsatisfiability of a system of word equations E, i.e., whether $E \in \mathbb{U}$.

Definition 4.28. A system of word equations E fulfils $E \in \mathbb{U}$ if an external solver decides $\not\models E$ or there exists an equation $\alpha \doteq \beta \in E$ such that at least one of the following cases holds:

1. There exists $\alpha_1 \in \operatorname{prefix}(\alpha) \cap A^*$ and $\beta_1 \in \operatorname{prefix}(\beta) \cap A^*$ such that $|\alpha_1| = |\beta_1|$ but $\alpha_1 \neq \beta_1$
2. There exists $\alpha_1 \in \operatorname{suffix}(\alpha) \cap A^*$ and $\beta_1 \in \operatorname{suffix}(\beta) \cap A^*$ such that $|\alpha_1| = |\beta_1|$ but $\alpha_1 \neq \beta_1$,
3. We have $|\alpha|_{\mathtt{x}_0} = |\beta|_{\mathtt{x}_0}$ for all $\mathtt{x}_0 \in \mathcal{X}$, but there exists a letter $a \in A$ such that $|\alpha|_a \neq |\beta|_a$.
4. if $\alpha \in A^*$ and there exists a word $\beta_1 \in \operatorname{factor}(\beta) \cap A^*$ and $\beta_1 \notin \operatorname{factor}(\alpha)$.

The first condition specifies a mismatch of terminals in the prefixes, whereas the second one characterises this behaviour for suffixes. The third case catches word equations where the amount of all variables is the same on both sides, but the letter-counts of some terminal in the two sides differ.

In all these cases we trivially cannot find a solution to the original system of word equations E. All conditions are in a sense variants of Lemma 4.14, adjusted to the current setting of the previously seen conditions.

The set of satisfiable system of word equations $\mathbb{S}$ is defined as follows.

Definition 4.29. A system of word equations E fulfils $E \in \mathbb{S}$ if $E = \emptyset$ or an external solver decides $\models E$.

In general the simplification rules $\rightsquigarrow_{\mathsf{s}}$ lead to the trivial satisfiable system of word equations $\emptyset$. Therefore, in our framework we let the trivially satisfied system of word equations $\emptyset$ be in $\mathbb{S}$. As our transformation rules guarantees that if $E_0 \in \mathsf{SAT}$, then there exists a sequence $E_0 \xrightarrow{r_0} \ldots \xrightarrow{r_{n-1}} \emptyset$, having $\mathbb{S} = \{\emptyset\}$. However, this sequence may be too long so we additionally let systems of word equations $E \in \mathbb{S}$ if an external solver decides $\models E$.

Remark 4.30 (Word Equations with Length Constraints). Whenever we reach a systems of word equations with length constraints $(\emptyset, L)$, i.e. an empty set of word equations with the system with length constraints L, we check if the length constraints are satisfiable. If they are, then $(E, L) \in \mathbb{S}$ otherwise $(E, L) \in \mathbb{U}$. In practice, we check the length constraints satisfiability with an external integer solver.

If the right hand side of a constraint $l \in L$ side evaluates to zero, we simply check whether $\emptyset \models l$ holds. This is the case if $\alpha \in A^*$ for a corresponding variable $\mathsf{x} \in \mathcal{X}$. If there is a linear constraint $l \in L$ such that $\not\models l$ the word equation system with linear constraints does not have a solution. Whenever applying a transition rule to the system of word equations, we modify all linear constraints $l \in L$ according to the r-function given in Definition 4.20. If $E' = \emptyset$, but $\not\models L'$, the equation system with linear constraints is not satisfiable. Therefore, $(E', L') \in \mathbb{U}$.

Checking the above mentioned condition on the two new equations might give us a quicker way of determining unsatisfiability of a word equation system. This allows us to refine the set $\mathbb{U}$. Consider for example the word equation $a\mathsf{x}_1\mathsf{x}_2b \doteq \mathsf{x}_1ba\mathsf{x}_2$. Simply applying the above rules does not lead to an unsatisfiable result. Applying s_3 first gives us the two equations $a\mathsf{x}_1 \doteq \mathsf{x}_1b$ and $\mathsf{x}_2b \doteq a\mathsf{x}_2$, where the first one is clearly unsatisfiable by the third condition given in Definition 4.28.

A system $(E, L) \subseteq \mathsf{WL}_A$ is also deemed unsatisfiable $((E, L) \in \mathbb{U})$ if there exists $(\sum_{\mathsf{x}_0 \in \mathcal{X}} c_{\mathsf{x}_0} \cdot \bar{\mathsf{x}_0} \leqslant r) \in L$ where for all $\mathsf{x}_0 \in \mathcal{X}$ we have $c_{\mathsf{x}_0} = 0$ and $r < 0$.

Example 4.31. Consider the systems of word equations with length constraints $(\emptyset, L)$ with $L = \{ 3 \cdot \bar{\mathsf{x}_0} \leqslant 3 \}$. Substituting $\bar{\mathsf{x}_0}$ by 0 immediately leads to $0 \leqslant 3$ and therefore $(\emptyset, L) \in \mathbb{S}$. Now consider the systems of word equations with length constraints $(\{ \mathsf{x}_0 a \doteq baa, \mathsf{x}_1 a \doteq a\mathsf{x}_0 \}, L)$. The computed substitution will map x_0 to ba giving us the system of word equations $E' = \{ \mathsf{x}_1 a \doteq aba \}$. The modifications to L accomodating for this transformation gives the set of linear length constraints $L' = \{0 \leqslant 3 - (3 \cdot 2)\} = \{0 \leqslant -3\}$. L' is obviously unsatisfiable and therefore $(E', L') \in \mathbb{U}$.

Example 4.32. Consider the terminal alphabet $A = \{ a, b, c, d, e \}$, the variable alphabet $\mathcal{X} = \{ \mathsf{x}_1, \mathsf{x}_0 \}$ and the systems of word equations with length constraints (E, L) with $E = \{\mathsf{x}_0 ada\mathsf{x}_0\mathsf{x}_1 \doteq cadace\mathsf{x}_0 bc\}$ and length constraints $L = \{\bar{\mathsf{x}}_0 \geqslant 1, \bar{\mathsf{x}}_1 \geqslant 3\}$. The graph induced by our transformations rules is depicted in Figure 4.5. Within this graph dark grey nodes indicate satisfiable systems of word equations, light grey nodes unsatisfiable systems of word equations, and white nodes unknown status. Starting at the initial systems of word equations with length constraints we are able to apply the rules $l_{4^{\mathtt{pre}}}$, $l_{4^{\mathtt{suf}}}$,$l_{5^{\mathtt{pre}}}$, and $l_{5^{\mathtt{suf}}}$, other rules are not applicable at that point due to the structure of the word equation and the corresponding length constraints. Nevertheless, the erasing rule directly leads to unsatisfiable system of word equations. Consider applying $l_{5^{\mathtt{pre}}}$ which leads to the word equation $ada\mathsf{x}_1 \doteq cadacebc$ and a partial substitution $r(\mathsf{x}_0) = \varepsilon$.

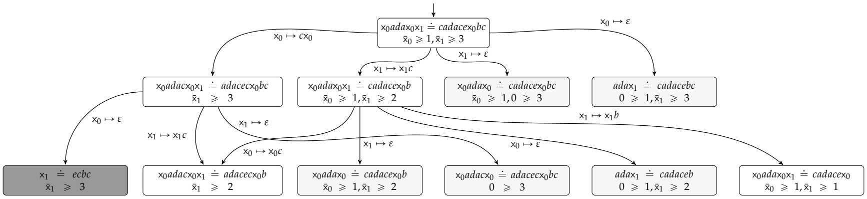

$\mathsf{x}_0 ada\mathsf{x}_0\mathsf{x}_1 \doteq cadace\mathsf{x}_0 bc$
$\bar{\mathsf{x}}_0 \geqslant 1, \bar{\mathsf{x}}_1 \geqslant 3$
$\mathsf{x}_0 \mapsto c\mathsf{x}_0$
$\mathsf{x}_0 \mapsto \varepsilon$
$\mathsf{x}_1 \mapsto \varepsilon$
$\mathsf{x}_1 \mapsto \mathsf{x}_1 c$
$\mathsf{x}_0 adac\mathsf{x}_0\mathsf{x}_1 \doteq adacec\mathsf{x}_0 bc$
$\bar{\mathsf{x}}_1 \geqslant 3$
$\mathsf{x}_0 ada\mathsf{x}_0\mathsf{x}_1 \doteq cadace\mathsf{x}_0 b$
$\bar{\mathsf{x}}_0 \geqslant 1, \bar{\mathsf{x}}_1 \geqslant 2$
$\mathsf{x}_0 ada\mathsf{x}_0 \doteq cadace\mathsf{x}_0 bc$
$\bar{\mathsf{x}}_0 \geqslant 1, 0 \geqslant 3$
$ada\mathsf{x}_1 \doteq cadacebc$
$0 \geqslant 1, \bar{\mathsf{x}}_1 \geqslant 3$
$\mathsf{x}_0 \mapsto \varepsilon$
$\mathsf{x}_1 \mapsto \mathsf{x}_1 c$
$\mathsf{x}_1 \mapsto \varepsilon$
$\mathsf{x}_0 \mapsto \mathsf{x}_0 c$
$\mathsf{x}_1 \mapsto \varepsilon$
$\mathsf{x}_0 \mapsto \varepsilon$
$\mathsf{x}_1 \mapsto \mathsf{x}_1 b$
$\mathsf{x}_1 \doteq ecbc$
$\bar{\mathsf{x}}_1 \geqslant 3$
$\mathsf{x}_0 adac\mathsf{x}_0\mathsf{x}_1 \doteq adacec\mathsf{x}_0 b$
$\bar{\mathsf{x}}_1 \geqslant 2$
$\mathsf{x}_0 ada\mathsf{x}_0 \doteq cadace\mathsf{x}_0 b$
$\bar{\mathsf{x}}_0 \geqslant 1, \bar{\mathsf{x}}_1 \geqslant 2$
$\mathsf{x}_0 adac\mathsf{x}_0 \doteq adacec\mathsf{x}_0 bc$
$0 \geqslant 3$
$ada\mathsf{x}_1 \doteq cadaceb$
$0 \geqslant 1, \bar{\mathsf{x}}_1 \geqslant 2$
$\mathsf{x}_0 ada\mathsf{x}_0\mathsf{x}_1 \doteq cadace\mathsf{x}_0$
$\bar{\mathsf{x}}_0 \geqslant 1, \bar{\mathsf{x}}_1 \geqslant 1$

Figure 4.5. Levis's graph for Example 4.32

Therefore, r violates the linear constraint $\bar{\mathtt{x}}_0 \geqslant 1$. The reasoning is exactly the same for rule $l_{5^{\mathtt{suf}}}$.

Continuing with rule $l_{4^{\mathtt{pre}}}$ leads to a word equation $c\mathtt{x}_0 adac\mathtt{x}_0\mathtt{x}_1 \doteq cadacec\mathtt{x}_0 bc$ and the partial substitution $r(\mathtt{x}_0) = c\mathtt{x}_0$. By applying the simplification rules (in this case $s_{1^{\mathtt{pre}}}$) we get the simplified word equation $\mathtt{x}_0 adac\mathtt{x}_0\mathtt{x}_1 \doteq adacec\mathtt{x}_0 bc$ without any modifications to the partial substitution r. According to r, the linear constraint $\bar{\mathtt{x}}_0 \geqslant 1$ is modified to $\bar{\mathtt{x}}_0 \geqslant 1 - 1 = 0$. Since this constraint does not restrict the possible solutions, it is not present in the successor systems of word equations with length constraints. Continuing with the analysis of this systems of word equations with length constraints we are able to apply either rule $l_{5^{\mathtt{pre}}}$, $l_{4^{\mathtt{pre}}}$, or $l_{4^{\mathtt{suf}}}$. Applying rule $l_{5^{\mathtt{pre}}}$ will lead to the systems of word equations with length constraints consisting of a word equation $adac\mathtt{x}_1 \doteq adacecbc$ with linear constraint $\bar{\mathtt{x}}_1 \geqslant 3$. The simplification rule $s_{1^{\mathtt{pre}}}$ first minimises the word equation to $\mathtt{x}_1 \doteq ecbc$. The rule $s_{4^{\mathtt{pre}}}$ simplifies further to a trivial satisfiable word equation $\varepsilon \doteq \varepsilon$ with solution $r(\mathtt{x}_1) = ecbc$. The modification of the linear constraint $\bar{\mathtt{x}}_1 \geqslant 3$ changes depending on r to $\bar{\mathtt{x}}_1 \geqslant 3 - 4 = -1$ and trivially evaluates to true. By backtracking this path, combining r functions, we get the solution $h = \{\, \mathtt{x}_0 \mapsto c, \mathtt{x}_1 \mapsto ecbc \,\}$.

In Figure 4.5 we visualise the whole layer of applicable transitions. However, in practice, we will terminate as soon as we found a suitable solution. We can try exploring the graph further by considering the white nodes within the lowest layer.

4.2.4 Implementation

The preceding sections presented our framework for solving systems of word equations. However, we left the algorithm underspecified by relying on non-determinism. In this section, we present how we extended Woorpje with this technique and, as part of this, we describe how the non-determinism of Algorithm 2 is managed. For the remainder of the section by Woorpje we refer to the algorithm presented here. Woorpje uses CVC4, Z3str3, and Z3seq as assisting string constraint solvers. The assisting SMT solver doubles as arithmetic solver for the check mentioned in Remark 4.30. In the theoretical discussion of our technique, we only mentioned using external string solvers, but never elaborated on *when* or *how* they are called. In regards to when, Woorpje can be configured to use one of five different heuristics for calling the external solver:

1. A predefined depth $d \in \mathbb{N}$ of the transformation system is reached.

2. In a system of word equations E each equation $e \in E$ has exceeded a bound $b \in \mathbb{N}$, i.e. $b \leqslant |e|$.

3. For a transformation between two system of word equations E and E' with an r-function $E \xrightarrow{r} E'$, the length of each word equation $e \in E$ increased too quickly, meaning that, for a predefined factor $s \in \mathbb{Q}$, we get $(\sum_{e \in E} |e|) \cdot s < (\sum_{e \in E'} |e|)$. This gives us an indicator to the presence of a long solution, where applying our rules might take longer.

4. While performing a transformation step $E \xrightarrow{r} E'$ the ratio between variables and terminals changes to an unfortunate extent w.r.t. our rules, meaning for $s \in \mathbb{Q}$, we have $t(E) = \left(\sum_{e \in E} |e_{|A}|\right)$ and $v(E) = \left(\sum_{e \in E} |e_{|\mathcal{X}}|\right)$ and $\frac{t(E)}{v(E)} \cdot s < \frac{t(E')}{v(E')}$. If this happens, we know about a massive increase of letters in the middle of at least one side of the word equation, thus it is not useful to apply our rules.

5. Never call the SMT solver. This is mainly useful for the evaluation of the heuristics, to obtain a baseline.

The first heuristic provides a mechanism to directly using the assisting solver at a specific depth of the graph induced by our transformation rules, meaning after visiting d successors of the initial system of word equations, we involve an external solver. It allows us to split the search after reaching a predefined level. Our empirical evaluation of this technique has shown, that the external solvers often struggle on doing the initial decisions. This heuristic takes care of these initial steps and involves the external solver at a later point of solving a system of word equations.

The second heuristic calls an external solver if a system of word equations reaches a certain length. Long word equations usually require many applications of our transformation rules. By using this heuristic, we might be able to avoid exploring this long transformation-path and maybe reach a solution faster.

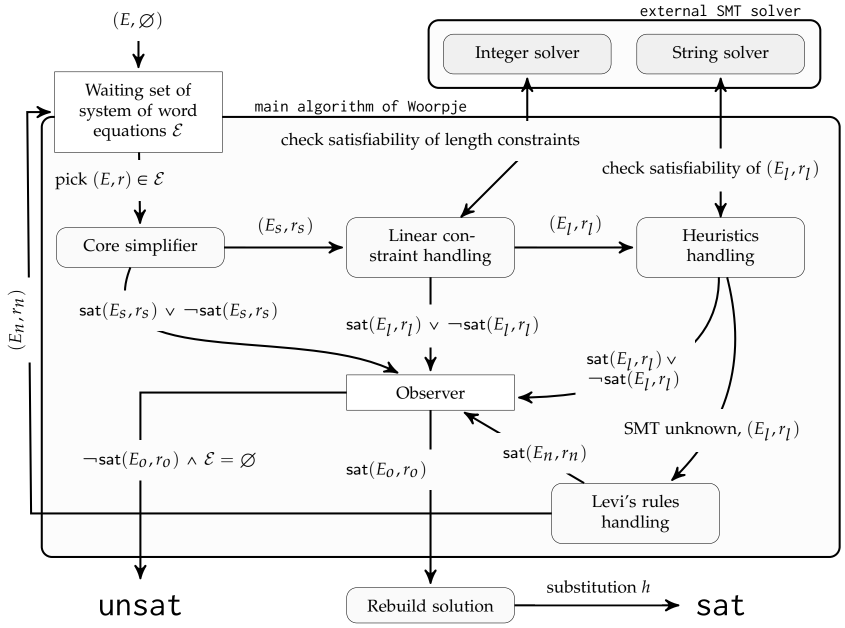
The third and the fourth heuristic target equations whose length are rapidly increasing. Consider for example the equation

$$\mathtt{x}_0 a \mathtt{x}_0 b \mathtt{x}_1 b \mathtt{x}_2 \doteq a \mathtt{x}_0 \mathtt{x}_1 \mathtt{x}_1 b \mathtt{x}_2 \mathtt{x}_2 baa.$$

This equation has a minimal solution $h = \{\mathtt{x}_0 \mapsto aaaaaaaa, \mathtt{x}_1 \mapsto aaaa, \mathtt{x}_2 \mapsto aa\}$ (the general form of this word equation (see Proposition 1 of [43]) has a minimal solution which is exponential in the length of the word equation). Solving this equation by only applying our transformation rules would potentially lead to an exponentially long chain of applications of $\mathtt{l}_{4^{\mathtt{pre}}}$ rules. By using appropriate parameters for the given heuristic we are able to prune the search with a call to an external solver.

In principle the external solver could run until it reaches a conclusion. This is, however, not necessarily a good idea. Sometimes WOORPJE can solve equations, using the transformation system, quicker than the time needed by the external solvers for reaching their conclusion. Therefore, WOORPJE gives the external solver a time limit (also affecting the arithmetic solver) within which it has to reach a verdict. In case the external solvers times out, WOORPJE simply puts the system of word equations into `Waiting`, and eventually reconsiders it.

The detailed implementation of algorithmic step 10 in Algorithm 2 is visualised in more detail in Figure 4.6. We randomly pick a system of word equations E from the `Waiting` set. Afterwards, within our core simplifier, we apply the rules defined by $\rightsquigarrow_{\mathtt{s}}$. Whenever these rules detect $E_s \in \mathbb{U}$ or $E_s \in \mathbb{S}$ the result is reported to an observer. Otherwise, the simplified system of word equations E_s is forwarded to the unit, which handles the linear constraints as described in Example 4.2.1. To check for general satisfiability of the provided length constraints, we use an external integer solver. Within the current setup, based on the users choice, either CVC4 or Z3. Again, whenever this unit detects $E_s \in \mathbb{U}$ or $E_s \in \mathbb{S}$ the result will be reported to the observer. The modified system of word equations E_l is then forwarded to the heuristics unit. This part of the algorithm triggers the external SMT solver, whenever a heuristic presented above is triggered. Within the given time limit the external solver tries to classify E_l. Success is reported to the heuristics unit and forwarded to the observer. In the next step Levi's rules, presented in Section 4.2.2,



$(E, \varnothing)$
external SMT solver
Integer solver
String solver
Waiting set of system of word equations $\mathcal{E}$
main algorithm of Woorpje
check satisfiability of length constraints
check satisfiability of (E_l, r_l)
pick $(E, r) \in \mathcal{E}$
Core simplifier
(E_S, r_S)
Linear constraint handling
(E_l, r_l)
Heuristics handling
(E_n, r_n)
$\mathsf{sat}(E_S, r_S) \vee \neg\mathsf{sat}(E_S, r_S)$
$\mathsf{sat}(E_l, r_l) \vee \neg\mathsf{sat}(E_l, r_l)$
$\mathsf{sat}(E_l, r_l) \vee \neg\mathsf{sat}(E_l, r_l)$
Observer
SMT unknown, (E_l, r_l)
$\neg\mathsf{sat}(E_O, r_O) \wedge \mathcal{E} = \varnothing$
$\mathsf{sat}(E_O, r_O)$
$\mathsf{sat}(E_n, r_n)$
Levi's rules handling
unsat
Rebuild solution
substitution h
sat

Figure 4.6. Architecture of WOORPJE

are applied to E_l. In order to get the Waiting set as small as possible we are applying $\rightarrow_{\mathrm{L}}$ only based on one word equation at a time rather than branching according to the transformation $\rightarrow_{\mathrm{L}}$ defined for all equations $e \in E_l$. By just choosing one word equation, and transforming the system according to the rules this equation induces, we do not restrict ourselves: we serialise the case distinctions and analyse the result, instead of applying rules simultaneously. This might lead to an earlier detection of dead-ends, but, in general, it leads to the same complete case analysis as considering all possible rules simultaneously. Whenever a derived system of word equations is trivially satisfiable, the result is reported to the observer. All not trivially unsatisfiable system of word equations are pushed into the Waiting set. The observer forms the core of the algorithm. If one of the units reports the existence of an unsatisfiable system of word equations, the observer reports `unsat` if `Waiting` does not hold any other system of word equations. If it receives a positive answer, the solution to the equation is rebuilt as seen in Example 4.19. WOORPJE reports `sat` and the solution in this case.

4.2.5 Related Work

There are many string solvers levering the rules induced by Levi's Lemma (cf. [30]) and also the fact of choosing solvers based on heuristics (cf. [25]) is not a novel approach. From our point of view the novelty of our approach is that we centre word equations and simply modify other constraints while solving the system of word equations. This allows us to efficiently explore the induced graph. Given this fact, we analyse the structure of the graph and identify word equations where using our rules solely, simply results in reaching a resource limit. This is where we add our heuristics to invoke other solvers, using other solving procedures. To the best of our knowledge, this type of interleaving has not been analysed before.

4.2.6 Conclusion

In this section we present a transformation-system-based approach to solve a subset of string constraints, or, in other words, word equations with length constraints. The method is implemented in the tool WOORPJE and initial tests indicate its value (see Section 6.3). The instances we are able to solve with this technique, and were not solved before using solvers such as Z3STR3 and CVC4, are evidence supporting our claim that implementing this approach into existing string solvers is beneficial. As such, it seems that our approach enriches in a non-trivial way the current landscape of string solving.

In the future, we aim to extend our approach to include regular constraints, making it applicable to industrial case studies. Initial experiments reveal that this extension can be done similarly to the addition of length constraints: the modification of regular constraints through our transformations is purely influenced by the movement inside our transformation-system. Furthermore, an extension to the identification of $\mathbb{U}$ systems is required. We believe the transformation system could work very well in collaboration with the existing string theory solver of SMT solvers, if structural properties of the system of word equations under which the SMT solver works could be identified. A natural step for our work is thus to identify such structural properties. Related to this is exploiting the structure of the transformation graph during the search for a solution.

4.3 Automata-based solving of regular expression membership constraints

As many of the theories discussed in this work, theories including regular expression membership predicates are an active topic of research, posing interesting questions. Solely considering the theory of regular expression membership predicates, an elegant proof of their decidability is given in [4]. The theory of word equations and regular expression membership predicates is known to be decidable [85]. It is not known if the satisfiability problem for string constraints involving all aforementioned theories is decidable or not. However, already in the presence of other simple and natural constraints, like string-number conversion as seen in Section 2.4, this problem becomes undecidable (cf. [41]).

Driven by practical relevance and the need of more efficient algorithms, we analysed 56993 string solving instances from industrial applications and solver developers containing regular expression membership predicates, which we will introduce in detail in Chapter 5, and identified numerous relevant sub-theories based around regular membership predicates. In particular, we identified theories which may have a string-number conversion predicate numstr, a string length function or string concatenation, and prove decidability, respectively undecidability, for certain sub-theories. The value of this theoretical analysis of present data is massive, since the sub-theory occurring the most within these benchmarks is actually PSPACE-complete, as we show within this section. Most notably, these results directly influenced the implementation of an algorithm implemented within Z3str3 showing superior performance compared to its competitors, as we showcase in Section 6.4. The algorithm itself was directly formed by the ideas we used in the proofs of the theorems presented in this section.

In the following subsections we show that the theory of complement-free-regular expression membership predicates with linear length constraints and concatenation is PSPACE-complete. Furthermore, if we additionally allow complement, we prove decidability and an $\mathsf{NSPACE}(f(n))$ lower bound, where $f(n)$ is a tetration $\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2^{cn}}}}}}}_{k \text{ times}} = 2 \uparrow^{k} (cn)$ whose height k depends on the number of stacked complements (and c is a constant). Continuing this trail, we prove PSPACE-completeness for the theory of complement-free regular expression membership predicates and a string-number conversion predicate, which naturally leads to decidability when considering complements. We show the corresponding lower bound in this case, too. At the opposite end of our spectrum, we show that the theory of regular expression membership predicates, linear length constraints, concatenation and string-number conversion is in fact undecidable.

To summarise, our analysis of the benchmarks not only revealed these theories, but also shows that most considered real-world string constraints actually fall into a decidable fragment. Out of 56993, about 51% lay in a decidable fragment. Only considering string constraints without word equations (30540 of 56993 instances), 26140 of these instances (85%) fall into a decidable fragment. Therefore, our theoretical analysis gives an intuition with respect to the performance of our solver.

4.3.1 Identification of Relevant Theories

During the development of an extension to cope with regular membership constraints within Z3str3RE, we analysed a huge set of over 100000 industrial influenced benchmarks and identified 22425 instances containing at least one regular expression membership constraint. This set includes instances from the AppScan [121], BanditFuzz [102], JOACO [110], Kaluza [99], Norn [4], Sloth [62], Stranger [120], and Z3str3-regression [23] benchmarks. Additionally, we generated 19979 benchmarks based on a collection of real-world regular membership queries collected by Loris D'Antoni from the University of Wisconsin, Madison, USA. Thirdly, we applied StringFuzz's [27] transformers to instances supplied by Amazon Web Services related to security policy validation to obtain roughly 15000 instances. All benchmark sets and their origin will be introduced in

Chapter 5.

We analysed the benchmarks according to their structure, as well as predicates and functions. We identified sets which contain string-number conversion, string concatenation, or linear length constraints over variables used within the regular expression membership predicate. The benchmarks contained combinations of these operations. The goal is now to group them into different first order logic theories.

The resulting first order logic theories.

The basis of the following theories is built by $\mathcal{A}_s$, the theory of simple regular expressions, which removes the complement operation of regular expressions from the theory $\mathcal{A}_e$ introduced in Section 2.2.3. The reasons why considering these restrictions is manifold: 1. removing the complement often allows us to prove an exact PSPACE-bound for these theories, 2. it seems that within real-world applications of string constraints the presence of stacked complements can be neglected (as we will see in the following analysis).

The vocabulary of simple regular expressions is given by

$$\mathcal{R}_s = \{\, \cdot//2, \cup//2, {}^{*}//1, \dot{\in}\,/2, \dot{\varnothing}, \dot{\varepsilon} \,\} \subseteq \mathcal{R}_e.$$

The axioms of this theory also stay the same except for the ones affecting the complement. Let $\mathtt{RegEx}_A$ denote the set of all regular expressions without complement. Naturally, this continues with our many-sorted $\mathcal{R}_s$-structure

$$\begin{aligned}\mathcal{A}_s = \{\, &\mathtt{RegEx}_A, A^*, \cdot^A, \dot{\varepsilon}^A, \cdot^{\mathtt{RegEx}_A}, \cup^{\mathtt{RegEx}_A},\\ &*^{\mathtt{RegEx}_A}, \dot{\varnothing}^{\mathtt{RegEx}_A}, \dot{\varepsilon}^{\mathtt{RegEx}_A}, \dot{\in}^{A\,\mathtt{RegEx}_A} \,\} \subseteq \mathcal{A}_e.\end{aligned}$$

Based on our extended regular expressions $\mathcal{R}_e$ and simple regular expressions $\mathcal{R}_s$, while categorising the benchmarks, we identified three important, (partially) disjoint theories, forming extensions of the aforementioned theories.

In practice solutions to variables are often restricted by linear inequalities ranging over the length of potential solutions. Therefore, a natural extension is adding a function to our our vocabularies allowing to reason about length. Let

$$\mathcal{R}_{il} = \mathcal{R}_i \cup \{ \mathbb{Z}, +//2, \leqslant/2, \dot{0}, \mathsf{len}//1 \}$$

be a vocabulary where $i \in \{ e, s \}$, being characterised by previously defined axioms and additionally the associativity and commutativity of $+//2$, the existence of a neutral element, and the requirement that $\leqslant$ be a total and monotonic ordering on our domain. The many-sorted $\mathcal{R}_{il}$-structure of *regular expressions with length* is defined by

$$\mathcal{A}_{\text{il}} = \mathcal{A}_i \cup \{ +^{\mathbb{Z}}, \leqslant^{\mathbb{Z}}, \dot{0}^{\mathbb{Z}}, \mathsf{len}^{A \to \mathbb{Z}} \},$$

where $+^{\mathbb{Z}}$,$\leqslant^{\mathbb{Z}}$ are defined as commonly used operations over $\mathbb{Z}$, $\dot{0}^{\mathbb{Z}} = 0 \in \mathbb{Z}$, and the length function $\mathsf{len}^{A \to \mathbb{Z}}$ defined in Section 2.3.

A second addition often occurring in real-world program analysis is a string-number conversion predicate. To this extend let

$$\mathcal{R}_{in} = \mathcal{R}_i \cup \{ \text{numstr}/2 \}$$

whereas $i \in \{ e, s, el, sl \}$ be a vocabulary. The axioms are derived from the corresponding base theory. The many-sorted $\mathcal{R}_{in}$-structure of *regular expressions with number conversation* is defined by

$$\mathcal{A}_{\text{in}} = \mathcal{A}_i \cup \left\{ \mathbb{N}, \text{numstr}^{\mathbb{N}\, A^*} \right\},$$

whereas $\text{numstr}^{\mathbb{N}\, A^*}$ is a relation, which holds for all positive integers $i \in \mathbb{N}$ and words $w \in \{ 0, 1 \}^*$ where w – possibly having leading zeros – is the binary representation of i, as formally defined in Section 2.4.

Naturally, not only in real-world applications it is interesting to ask whether a pattern $\alpha \in \mathtt{Pat}_A$ possibly containing variables is bound by a regular language. This leads to the last extension we are considering in this section. Let

$$\mathcal{R}_{ic} = \mathcal{R}_i \cup \{ \cdot//2 \}$$

whereas $i \in \{ e, s, el, sl, eln, sln, en, sn \}$ be a vocabulary, having the additional axioms induced by $(\mathtt{Pat}_A, \cdot^A, \varepsilon)$ forming a monoid. The many-sorted $\mathcal{R}_{ic}$-structure of *regular expressions with concatenation* is defined by

$$\mathcal{A}_{\text{ic}} = \mathcal{A}_i \cup \left\{ \cdot^A, \dot{\varepsilon}^A \right\},$$

whereas $\cdot^A$ is defined as the classical concatenation over $\mathtt{Pat}_A$ and $\dot{\varepsilon}^A =$

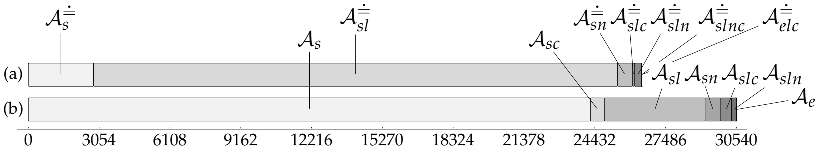


Figure 4.7. Distribution of instances among their theories. (a) instances with word equations (b) instances without word equations.

$\varepsilon \in A^*$. These theories are again naturally combined with the theory of word equations by simply considering the union of their components.

The following example gives an intuition with respect to the theories.

Example 4.33. Consider the string constraint $C = \mathsf{x}_1 \dot{\in} 1^* \wedge \text{numstr}(15, \mathsf{x}_1) \wedge \mathsf{len}(\mathsf{x}_1) \geqslant 3$ where $\mathsf{x}_1 \in \mathcal{X}$ and $1 \in A$. A solution $h \in \mathcal{H}_A$ is given by $h(\mathsf{x}_1) = 1111$, since $h(\mathsf{x}_1) = 1111 \in L(1^*)$, $\text{numstr}(15, 1111)$ because 1111 is the binary representation of 15, and $h(\mathsf{x}_1) \geqslant 3$. Therefore $\mathcal{A}_{sln}, h \models C$.

Throughout this section to ease readability, we write $\alpha \dot{\notin} R$ instead of $\neg(\alpha \dot{\in} R)$ for $\alpha \in \mathtt{Pat}_A$ and $R \in \mathsf{Reg}_A$.

Benchmark analysis

The analysis of the 56993 instances reveals that 30540 instances are solely a member of one of our regular expression theories, while 26453 additionally contained word equations. In Figure 5.7 we plot the distribution of all instances with respect to their theory. We display the instances according to the presence of word equations into two bars (a) and (b). The width of a single block within a bar corresponds to the instance count of the smallest theory. Since some of the theories are disjoint (e.g. $\mathcal{A}_{sl}$ and $\mathcal{A}_{sn}$) the diagram does not visualise inclusions.

Within formulae only containing regular membership constraints, the most frequented theory is $\mathcal{A}_s$ holding 24256 instances. As we will see in this work, this theory and also its successor $\mathcal{A}_{sl}$ with 4327 instances are PSPACE-complete and raises hope for efficient solving strategies. The theories $\mathcal{A}_{elnc}$ and $\mathcal{A}_{slnc}$, for which we prove undecidability within this work, do not seem to have a high relevance in application since they do not occur at all within our analysed set of benchmarks.

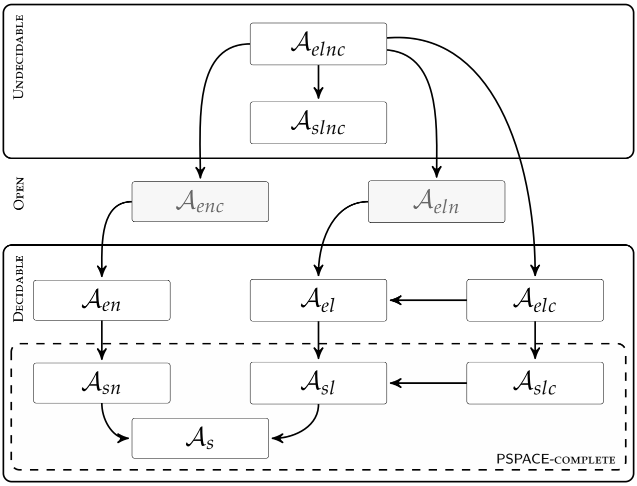


Figure 4.8. Visualization of relationship and decidability of various extensions of $\mathcal{A}_s$, with arrows leading from stronger theories to theories which they contain.

On the other hand, the instances containing word equations are also based around simple regular expressions. The most prominent theory is $\mathcal{A}_{sl}^{\doteq}$ holding 22604 instances, followed by $\mathcal{A}_{s}^{\doteq}$ containing 2813 instances. Unfortunately, the decidability of the largest set of instances is not known. Notably, the total set only contains nine instances based on the theory $\mathcal{A}_e$ where stacked complements are actually needed. All other instances can be rewritten to simply avoid the complement.

4.3.2 Embedding of the Discussed Theories

In this section, we characterise the related quantifier-free first-order theories introduced in Section 4.3.1 according to their decidability. The contributions are summarised in Figure 4.8. The arrows lead from stronger and more expressive theories to weaker ones. Theories in the upper box are undecidable, while those in the lower box are decidable (similarly, the theories within the inner dashed box are PSPACE-complete). We proceed with a summary of the theorems we prove and some discussion of the motivation and intuition for the proofs.

In an attempt to move from simpler to more complicated theories, we will begin our journey with the theory without complement operation for regular expressions. We will start be considering $\mathcal{A}_{slc}$. The motivation for approaching this theory first (formalized later in Theorem 4.39) is that for more general theories, which include regular expressions with complement operations, even simple tasks (like checking whether there exists a common string in the languages of two given expressions) require an exponential amount of space. One way to understand this is that the exponential blow-up with respect to the size of the regular expressions comes from transforming this expression into an NFA, determinising it, and then computing its complement. In fact, we will see that any other approach inherently leads to such an exponential blow-up. We can state the following result.

Lemma 4.34. *The satisfiability problems for $\mathcal{A}_{slc}$ and $\mathcal{A}_{sl}$ of simple regular expressions, linear integer arithmetic, string length, and concatenation are decidable in* PSPACE.

Proof. We first show this result for $\mathcal{A}_{slc}$. Consider a formula φ from $\mathcal{A}_{slc}$. We will give a non-deterministic algorithm that decides whether φ is satisfiable in polynomial space. However, for a simpler presentation, we first discuss an algorithm deciding the satisfiability of φ without the space restriction.

Firstly, we convert the formula φ into an equivalent formula φ' in negation normal form. Therefore, φ' consists only of a Boolean combination ($\vee$ and $\wedge$) of atoms of the form $\alpha \mathrel{\dot\in} R$ or $\alpha \mathrel{\dot\notin} R$, where $\alpha \in \mathtt{Pat}_A$ and $R \in \mathtt{RegEx}_A$, as well as atoms encoding arithmetic constraints. Clearly, $|\varphi'| \in \mathcal{O}(|\varphi|)$.

Secondly, we non-deterministically choose an assignment of truth values for all atoms such that the Boolean abstraction of φ' is satisfiable. As such, we get from our formula a list $\mathcal{L}_r$ of atoms of the form $\alpha \mathrel{\dot\in} R$ or $\alpha \mathrel{\dot\notin} R$, where $\alpha \in \mathtt{Pat}_A$ and $R \in \mathtt{RegEx}_A$, that have to evaluate to true; if an atom $\alpha \mathrel{\dot\in} R$ was assigned false in the assignment we chose, then we put $\alpha \mathrel{\dot\notin} R$ in the list, and if $\alpha \mathrel{\dot\notin} R$ was assigned false in our assignment, then we put $\alpha \mathrel{\dot\in} R$ in the list, while all the atoms that are assigned true are added to the list as they are. We similarly construct a second list $\mathcal{L}_l$ containing a set of arithmetic linear constraints that should be evaluated to true. If, and only if, we find an assignment of the variables occurring in these two lists such that all the atoms they contain are evaluated to true, φ is satisfiable.

Thirdly, if $\alpha \dot{\in} R$ is a regular membership constraint of $\mathcal{L}_r$, let M_R be an NFA such that $L(R) = L(M_R)$. Solving the constraint $\alpha \dot{\in} R$ is equivalent to solving $\alpha \in L(M_R)$. If $\alpha \dot{\notin} R$ is in $\mathcal{L}_r$, let $\overline{M}_R$ be an NFA such that $L(\overline{R}) = L(\overline{M_R})$, since solving the constraint $\alpha \dot{\notin} R$ is equivalent to solving $\alpha \in L(\overline{M}_R)$. Essentially, the list $\mathcal{L}_r$ can be seen as a list of constraints $\alpha \in L(M)$, where $\alpha \in \mathtt{Pat}_A$ and M is an NFA. Without loss of generality, we assume each of the NFAs appearing in $\mathcal{L}_r$ has exactly one initial state, one final state, and no ε-transitions.

Now, consider the constraint $\alpha \dot{\in} L(M)$ for $M = (Q, A, \delta, q_0, \{ f \})$ from our list, and note that α is either a single string variable or the concatenation of several string variables. It is clear that if $\models \alpha \dot{\in} L(M)$ there is a way of building an assignment $h \in \mathcal{H}_A$ such that for each variable $\mathsf{x} \in \mathtt{vars}\ (\alpha)$ there exists a path $h(\mathsf{x})$ in M and the entire pattern $h(\alpha)$ forms a path leading from q_0 to f in M. Therefore, $h(\alpha) \in L(M)$.

Let $\alpha = \mathsf{x}_1 \dots \mathsf{x}_k$ for $k \in \mathbb{N}$ and $\mathsf{x}_i \in \mathtt{vars}\ (\alpha)$. We non-deterministically choose a starting state $q_{\mathsf{x},i} \in Q$ and a final state $f_{\mathsf{x},i} \in Q$ for each occurrence $i \leqslant k$ of each variable $\mathsf{x} \in \mathtt{vars}\ (\alpha)$, such that there exists a $w_{\mathsf{x},i} \in A^*$ and $\delta(q_{x,i}, w) \subseteq \{ f_{x,i} \}$ and $\delta(q_0, w_{\mathsf{x}_1,1} \dots w_{\mathsf{x}_k,i}) \subseteq \{ f \}$ for $i = |\alpha|_{\mathsf{x}_k}$.

Each variable $\mathsf{x} \in \mathtt{vars}\ (\alpha)$ must have an assignment that is accepted by all NFAs $M_{\mathsf{x},j}$, constructed for each of its occurrences $j \leqslant \ell_{\mathsf{x}} = \sum_{(\alpha \dot{\in} R) \in \mathcal{L}_r} |\alpha|_x$ from each constraint $\alpha \in L(M)$. Hence, we intersect all NFAs $M_{\mathsf{x},j}$ for all x and all $j \in [\ell_{\mathsf{x}}]$ and get a new NFA $A_{\mathsf{x}} = (Q_{\mathsf{x}}, A, \delta_{\mathsf{x}}, q_{0_{\mathsf{x}}}, F_{\mathsf{x}})$.

Further, let $B_{\mathsf{x}} = (Q_{\mathsf{x}}, \{ a \}, \delta'_{\mathsf{x}}, q_{0_{\mathsf{x}}}, F_{\mathsf{x}})$ be the unary NFA obtained by re-labelling all transitions in A_{x} with a single letter a, namely $\delta'_{\mathsf{x}} = \{(q, a) \mapsto p \mid q, p \in Q, b \in A, \delta(q, b) \subseteq \{ p \}\}$. It is clear that the paths of A_{x} correspond bijectively to the paths of B_{x}. Let $m = |Q_{\mathsf{x}}|$ be the number of states of A_{x} and B_{x}. A well known result, related to the Chrobak normal form of unary automata [34], is that *all accepting paths* in B_{x} that go through a state $q \in Q_{\mathsf{x}}$ from the initial state $q_{0_{\mathsf{x}}}$ of B_{x} to the final state f_{x} of B_{x} can be succinctly represented as the shortest path from $q_{0_{\mathsf{x}}}$ going through q to f_{x}, whose length is $\ell_p^q \leqslant 2m$, and the shortest cycle containing q, whose length is $\ell_c^q \leqslant m$ (see, for instance, the statement and proof of Lemma 1 of [55]). Thus, for each state q of B_{x} (or, equivalently, A_{x}), we can find the smallest $\ell_c^q \leqslant 2m$ such that there is a path from $q_{0_{\mathsf{x}}}$ going through q to f_{x}

and the smallest $\ell_p^q \leqslant m$ such that there is a cycle containing q of length ℓ_p^q. Then, we get that all accepting paths going through q in B_{x} as well as in A_{x} (and consequently, all the corresponding words) have lengths of the form $\ell_p^q + r\ell_c^q$, for $r \in \mathbb{N}$. Conversely, there exists an accepting path in B_{x} going through q of length $\ell_p^q + r\ell_c^q$ for all $r \in \mathbb{N}$, and a word $w \in A$ such that $|w| = \ell_p^q + r\ell_c^q$ and $w \in L(A_{\mathsf{x}})$. This means that for each variable x we get a disjunction of length constraints of the form $\mathsf{len}(\mathsf{x}) = \ell_p^q + r\ell_c^q$ for some state $q \in Q_{\mathsf{x}}$ and $r \in \mathbb{N}$. We add the length restrictions obtained for each variable x occurring in each constraint within the list $\mathcal{L}_r$ to the list $\mathcal{L}_l$.

It remains to check whether the linear arithmetic constraints of $\mathcal{L}_l$ are satisfiable, which is decidable (see [36]). If all arithmetic constraints are satisfied, it automatically means that there exists an assignment for each variable x such that the regular membership constraints are satisfied, too. So, φ is satisfiable.

The above non-deterministic algorithm is clearly sound and terminates, but it does not run in polynomial space. There are several steps where we obviously may use exponential space; for instance, computing $\overline{M_R}$ from M_R or computing the intersection automaton A_{x}. Also, deciding the satisfiability of $\mathcal{L}_l$ can be done in polynomial space w.r.t. $|\varphi|$ if all the coefficients of the linear constraints in $\mathcal{L}_l$ can be represented in a number of bits polynomial in $|\varphi|$; thus, we need to show that this holds.

We will now explain how to implement the algorithm above in polynomial space. The first difference occurs when switching from regular expressions to automata in the list $\mathcal{L}_r$. If $\alpha \mathrel{\dot{\in}} R$ or $\alpha \mathrel{\dot{\notin}} R$ is a regular membership constraint of $\mathcal{L}_r$, let M_R be an NFA such that $L(R) = L(M_R)$. Clearly, obtaining an appropriate NFA can be carried out in polynomial time (cf. [57]). The constraint $\alpha \mathrel{\dot{\in}} R$ is equivalent to $\alpha \mathrel{\dot{\in}} L(M_R)$, while $\alpha \mathrel{\dot{\notin}} R$ is equivalent to $\alpha \mathrel{\dot{\notin}} L(M_R)$. So, in this implementation, the list $\mathcal{L}_r$ is seen as a list of constraints $\alpha \mathrel{\dot{\in}} L(M)$ or $\alpha \mathrel{\dot{\notin}} L(M)$, where $\alpha \in \mathtt{Pat}_A$ and M is an NFA. In this way, we avoid constructing the automaton $\overline{M_R}$ for any regular expression $R \in \mathtt{RegEx}_A$. We do not need to construct $\overline{M_R}$, as we can simulate it using M_R.

Let $M_R = (Q, A, \delta, q_0, F)$ be an NFA. If we want to construct the DFA D_R (e.g. using the powerset construction) and then compute the complement DFA $\overline{D}_R$, we would get that the states of $\overline{D}_R$ are tuples of (at most $|Q|$) states of M_R, the transitions are the transitions of M_R applied on the components of the tuples, and a tuple $(q_1, \ldots, q_\ell) \in Q^\ell$ for some $\ell \in [|Q|]$ is final if and only if $q_i \notin F$ for $i \in [\ell]$. Similarly, instead of working directly with the NFAs M_R from constraints $\alpha \in L(M_R)$, we will work with the corresponding DFAs D_R. Again, we simulate them, because we know that their states are tuples of states from M_R, and we can simulate their transitions by executing the transitions of M_R on the components of the tuples. Consequently, we only store the current position within the simulation of an automaton D_R.

Following the strategy above, we cannot explicitly construct the NFA A_x for a variable x. On the one hand, we have not obtained all the automata we intersected in order to construct A_x, and on the other hand, as a variable might have $\mathcal{O}(|\varphi|)$ occurrences and the NFAs associated with its occurrences have $\mathcal{O}(|\varphi|)$ states, the automaton A_x may be of exponential size. But even if we cannot effectively build A_x, we can simulate it. In the previous construction, A_x was obtained as the intersection of the NFAs corresponding to the occurrences of the variable x. We will now construct A_x as an intersection of DFAs, so it will also be deterministic. More precisely, its states are tuples of states corresponding to all automata $D_{R_{\mathsf{x},i}}$ for any occurrence x within our simulation. In such a tuple, the position corresponding to an occurrence of x in a constraint $\alpha \in L(M_R)$ stores a state of D_R, which can be seen as a tuple of states of M_R. The position corresponding to an occurrence of x in a constraint $\alpha \notin L(M_R)$ stores a state of D_R, so, once more, a tuple of states of M_R. The transitions in A_x can be simulated by executing the transitions on components, using the corresponding automata, and a state of A_x is final if all its components are final. Clearly, the size of each state of A_x (i.e., the number of elements in each tuple representing a state) is $\mathcal{O}(|\varphi|^2)$. The states, final states, and transitions of B_x are the same as the ones of A_x. The value of m, the number of states of B_x and A_x, is $\mathcal{O}(|\varphi|^{|\varphi|^2})$. Note that m can be represented with $\mathcal{O}(|\varphi|^2 \log |\varphi|)$ bits. Also, note that, in this implementation, A_x is deterministic, while B_x is not, since all transitions now have the same label.

Finally, we need to show how the assignment of each variable x is done. We non-deterministically choose a state of A_x (and B_x) such that the word assigned to x labels a path that must go through q. Then we non-deterministically guess the corresponding ℓ_p^q and ℓ_c^q (whose value can be represented on a polynomial number of bits, according to the upper bounds $\ell_p^q \leqslant 2m$ and $\ell_c^q \leqslant m$) and check if there is a path of length ℓ_p^q from the initial state through q to the final state, and a cycle of length ℓ_c^q containing q. Finally, we add the constraint $\mathsf{len}(\mathsf{x}) = \ell_p^q + r\ell_c^q$ for some state $q \in Q_{\mathsf{x}}$ and $r \in \mathbb{N}$ to $\mathcal{L}_l$. We do this for all variables.

It can be shown by standard methods (cf. [36]) that if the integer program defined by $\mathcal{L}_l$ has a solution, then there is a solution contained inside the sphere of radius $c^{|\varphi|\cdot L}$ centred in the origin, where c is a constant and L is polynomial in the total number of bits needed to write the coefficients of the constraints in $\mathcal{L}_l$. So, it is enough to look for a solution to $\mathcal{L}_l$ inside the sphere of radius $c^{|\varphi|\cdot L}$. This means that the value of each variable from $\mathcal{L}_l$ can be written (in binary) in a polynomial number of bits. It is enough to guess an assignment for these variables (which can be stored in polynomial space) and then check if this is a solution to the integer programming problem (which can be done polynomial space). If the guess was correct, then φ is satisfiable.

This modified implementation now runs in polynomial space, so this concludes the proof for $\mathcal{A}_{slc}$. The result for $\mathcal{A}_{sl}$ follows immediately.

This ends the proof of the upper bound. □

Lemma 4.35. *The satisfiability problems for $\mathcal{A}_{slc}$ and $\mathcal{A}_{sl}$ of simple regular expressions, linear integer arithmetic, string length, and concatenation are* PSPACE*-hard.*

Proof. The following problem is PSPACE-complete [60, 74] – no matter whether the regular languages are given as regular expressions, deterministic, or non-deterministic finite automata.

> Let $L_1, \ldots, L_n$ be n regular languages over an alphabet A for $n \in \mathbb{N}$. Decide whether there exits a word $\alpha \in A^*$ such that
>
> $$\alpha \in \bigcap_{1 \leqslant i \leqslant n} L_i.$$

This problem can be reduced to the satisfiability problem for the quantifier-free theory $\mathcal{A}_{sl}$. Let $R_i \in \mathtt{RegEx}_A$ be a regular expression such that $L(R_i) =$

L_i, for all $i \in \{1, \ldots, n\}$. We now define a formula

$$\varphi = \bigwedge_{1 \leqslant i \leqslant n} \mathsf{x} \mathrel{\dot{\in}} R_i$$

where $\mathsf{x} \in \mathcal{X}$. Clearly, φ is satisfiable if and only if

$$\bigcap_{1 \leqslant i \leqslant n} L_i \neq \varnothing.$$

This ends the proof of the lower bound. □

Theorem 4.36. *The satisfiability problems for $\mathcal{A}_{slc}$ and $\mathcal{A}_{sl}$ of simple regular expressions, linear integer arithmetic, string length, and concatenation are* PSPACE*-complete.*

Proof. Immediate consequence of Lemma 4.34 and Lemma 4.35. □

When we allow arbitrary complements in the regular expressions, we can still prove the decidability of the respective theories but the complexity increases.

Theorem 4.37. *The satisfiability problems for $\mathcal{A}_{elc}$ and $\mathcal{A}_{el}$ of regular expressions, linear integer arithmetic, concatenation, and string length are decidable.*

Proof. The idea is to use the same strategy as explained above for $\mathcal{A}_{slc}$. Since regular expressions may now contain complements, when constructing the automaton M_R associated with a regular expression $R \in \mathtt{RegEx}_A$ we might have an exponential blow-up in size, even if the alphabet of the regular expression (resp. NFA) is binary and only one complement is used (as shown, for instance, in [63]). We can no longer guarantee the polynomial space complexity of our approach, but the decidability result holds. □

The last theorem is supplemented by the following remark, which shows upper and, more interestingly, lower bounds for the space needed to decide the satisfiability problem for a formula in the quantifier-free theories $\mathcal{A}_{el}$ and $\mathcal{A}_{elc}$.

Remark 4.38. Let $g : \mathbb{N}_{>0} \times \mathbb{Q} \to \mathbb{Q}$ recursively defined by $g(1, c) = 2^c$ and $g(k+1, c) = 2^{g(k,c)}$ for $k \in \mathbb{N}_{>0}$ and $c \in \mathbb{Q}$. Informally this mapping corresponds to the following tower of powers (a.k.a. tetration)

$$g(k, c) = \underbrace{2^{2^{2^{\cdots^{2^c}}}}}_{k \text{ times}} = 2 \uparrow^k c.$$

For a regular expression $R \in \mathtt{RegExC}_A$, define the complement-depth $\mathsf{cDepth} : \mathtt{RegExC}_A \to \mathbb{N}$ recursively as follows. If $R \in \{ \varnothing, \varepsilon, a \}$ for $a \in A$ let $\mathsf{cDepth}(R) = 0$. Otherwise if $R \in \{ R_1 \cup R_2, R_1 \cdot R_2 \}$ let $\mathsf{cDepth}(R) = \mathsf{cDepth}(R_1) + \mathsf{cDepth}(R_2)$, if $R = R_1^*$ let $\mathsf{cDepth}(R) = \mathsf{cDepth}(R_1)$, and if $R = \overline{R_1}$ let $\mathsf{cDepth}(R) = 1 + \mathsf{cDepth}(R_1)$ for appropriate $R_1, R_2 \in \mathtt{RegExC}_A$. For a formula φ in the quantifier-free theory $\mathcal{A}_{elc}$ (as well as $\mathcal{A}_{el}$) we let $\mathsf{cDepth}(\varphi)$ be the maximum depth of a regular expression in φ.

One can show, using for instance our approach from the proofs of Theorems 4.36 and 4.37, that the satisfiability problem for formulae φ from the quantifier-free theory $\mathcal{A}_{elc}$ (and $\mathcal{A}_{el}$ as well), with size $n \in \mathbb{N}$ and $\mathsf{cDepth}(\varphi) = k \in \mathbb{N}$, is in $\mathsf{NSPACE}(f(g(k-1, 2n)))$, where f is a polynomial function. However, there exists a positive rational number $c \in \mathbb{Q}$ such that the respective problem is not contained in $\mathsf{NSPACE}(g(k-1, cn))$. This lower bound follows from [107]. There, the following problem is considered: Given a regular expression $R \in \mathtt{RegExC}_A$, of length n, with $\mathsf{cDepth}(R) = k \in \mathbb{N}$ over an alphabet A, decide whether $L(R) = A^*$. It is shown that there exists a positive rational number $c \in \mathbb{Q}$ such that the respective problem cannot be solved in $\mathsf{NSPACE}(g(k, cn))$. So, deciding whether a formula φ of $\mathcal{A}_{el}$ consisting of the atoms $\alpha \mathrel{\dot\in} \overline{R}$ and $\alpha \in A^*$, where $R \in \mathtt{RegExC}_A$ is a regular expression of length n with $\mathsf{cDepth}(R) = k - 1$, is not contained in $\mathsf{NSPACE}(g(k-1, cn))$ (note that, in this case, the length of the formula φ is also $\mathcal{O}(n)$).

Intuitively, this lower bound shows that if the complement-depth of a formula of length n is k, then checking its satisfiability inherently requires an amount of space proportional to the value of the exponentiation tower of height $k - 1$, and with the highest exponent cn. ◁

Let g be defined as given in Remark 4.38. Based on the classical results from [107], we can derive the following theorem.

Theorem 4.39. *There exists a positive rational number $c \in \mathbb{Q}$ such that the satisfiability problem for the fragments of $\mathcal{A}_{el}$ and $\mathcal{A}_{elc}$ allowing only formulae of complement-depth at least k is not in* $\mathsf{NSPACE}(g(k-1, cn))$.

Proof. This is a direct consequence of Remark 4.38. □

This theorem shows that, in fact, when deciding the satisfiability problem for the quantifier-free theories $\mathcal{A}_{elc}$ and $\mathcal{A}_{el}$ the automata-based proof we presented is relatively close to the space-complexity lower bound for this problem. Any other approach, automata-based or otherwise, would still face the same obstacle: the space complexity of any algorithm deciding the satisfiability of formulae of complement-depth k cannot go under the $\mathsf{NSPACE}(g(k-1, cn))$ bound. This, on the one hand, explains our interest in analysing the theory $\mathcal{A}_{sl}$ (and its variants): as soon as we consider stacked complements, we are out of the PSPACE complexity class. On the other hand, this also explains the reason why in developing a practical solution for the satisfiability problem of $\mathcal{A}_{el}$ formulae within our tool Z3str3RE we use many heuristics. While the result of Theorem 4.37 was known from [82], our approach seems to provide a deeper understanding of the hardness of this problem, where this stems from, and of the ways we can deal with it.

Next we consider the case of replacing the length function by a string-number predicate. The lower bound of Theorem 4.39 applies also to the case of $\mathcal{A}_{en}$. So one cannot hope to solve the satisfiability problem for this theory in polynomial space, as soon as we allow arbitrary complements in our regular expressions. However, we can show that the satisfiability problem for $\mathcal{A}_{en}$ is decidable, and in PSPACE when only simple regular expressions are allowed.

Theorem 4.40. *The satisfiability problem for $\mathcal{A}_{sn}$ (resp. for $\mathcal{A}_{en}$) of (simple) regular expressions and a string-number predicate is* PSPACE*-complete (resp. decidable).*

Proof. The lower bound follows as in Theorem 4.36. We now show the PSPACE upper bound.

Let φ be a formula of length $n \in \mathbb{N}$ in the theory $\mathcal{A}_{sn}$. Firstly, once more, convert φ into an equivalent formula φ' in negation normal form which consists of a Boolean combination of atoms of the form $\alpha \dot{\in} R$ or $\alpha \dot{\notin} R$, where $\alpha \in \mathtt{Pat}_A$ and $R \in \mathtt{RegEx}_A$ (thus R is not containing any complement), as well as atoms encoding arithmetic constraints, and string-number predicates. Clearly, $|\varphi'| \in \mathcal{O}(|\varphi|)$.

Similar to the proof of Lemma 4.34, secondly, we non-deterministically guess the truth assignment of all atoms (regular constraints, arithmetic constraints, or string-number predicates) such that φ' evaluates to true. We can construct a list $\mathcal{L}_r$ of atoms of the form $\alpha \dot{\in} R$ or $\alpha \dot{\notin} R$, where α is a string term and R is a simple regular expression, that all have to evaluate to true. We also construct a second list $\mathcal{L}_l$ containing a set of arithmetic linear constraints that should all be true.

Thirdly, we process the string-number conversion predicates of the form $\text{numstr}(m, \alpha)$ and $\neg\text{numstr}(m, \alpha)$, where m is an integer term and $\alpha \in A^* \cup \mathcal{X}$. Note, since we do not allow concatenation, α can only be a word consisting of constants or a single variable. If m is neither a variable nor a constant, we add a new integer variable x_m and replace $\text{numstr}(m, \alpha)$ (respectively, $\neg\text{numstr}(m, \alpha)$) by the predicate $\text{numstr}(\mathsf{x}_m, \alpha)$ (respectively, $\neg\text{numstr}(\mathsf{x}_m, \alpha)$) and the arithmetic atom $\mathsf{x}_m = m$. A similar processing can be done to replace the constant strings from string-number predicates by variables. In this way, we obtain a new formula φ'', still of size $\mathcal{O}(|\varphi|)$. After this, each term in every numstr predicate is either a constant or variable of the appropriate sort.

Now, in φ'', if we have a string-number predicate $\text{numstr}(m, \alpha)$ (respectively, $\neg\text{numstr}(m, \alpha)$) where $m \in \mathbb{Z}$ is a constant, we let $\mathsf{bin}(m)$ be the constant string consisting of the shortest binary representation of m. We add $\alpha \dot{\in} 0^*\mathsf{bin}(m)$ (respectively, $\alpha \dot{\notin} 0^*\mathsf{bin}(m)$) to the list of regular constraints $\mathcal{L}_r$. We remove $\text{numstr}(m, \alpha)$ (respectively, $\neg\text{numstr}(m, \alpha)$) from φ''. If we have $\text{numstr}(\mathsf{x}, \alpha)$ (respectively, $\neg\text{numstr}(\mathsf{x}, \alpha)$) where x is an integer variable, we add $\alpha \dot{\in} 0^*(0,1)^*$ (respectively, $\alpha \dot{\notin} 0^*(0,1)^*$) to the regular constraints $\mathcal{L}_r$. We remove $\text{numstr}(\mathsf{x}, \alpha)$ (respectively, $\neg\text{numstr}(\mathsf{x}, \alpha)$) from φ'', but store in a new list $\mathcal{L}_b$ the information that the binary representation of x fulfils the same regular constraints as α (e.g., if we have $\alpha \dot{\in} R$ we add $\mathsf{x} \dot{\in} R$ as well), or, respectively, the complement of the regular constraints of α. In the latter case, it is worth noting that if we have a restriction $\alpha \dot{\notin} R$, the binary representation of x must be in the language defined by R, so we will not obtain regular expressions with stacked complements.

In this way we obtain a list of regular constraints that need to be true, a list of arithmetic linear constraints that need to be true, as well as a list of constraints stating that the binary representation of certain integer variables must also fulfil the same regular constraints as certain variables.

So far, all transformations can be clearly carried out in polynomial space with respect to $|\varphi|$. So, in $\mathcal{L}_l$, all coefficients can be represented in a polynomial number of bits, by the same reasons as before. Let $s \in \mathbb{Z}$ be the sum of the absolute values of all the constants occurring in the arithmetic constraints. Clearly s can be represented in a number of bits polynomial in $|\varphi|$. The list $\mathcal{L}_b$ remains unchanged.

Unlike the algorithm presented in Lemma 4.34, we will not solve the integer linear system defined by $\mathcal{L}_l$ using integer programming tools. Instead, we use the fact that deciding whether the set of linear constraints is satisfiable is equivalent to checking whether the language accepted by a finite synchronized multi-tape automaton A is empty or not (cf. [54]). This automaton has as states p-tuples of integers ranging over the set $\{\, i \in \mathbb{Z} \mid -\ell \leqslant i \leqslant \ell \,\}^p$ for an appropriate $\ell \in \mathbb{Z}$, where $p \in \mathbb{N}$ is the number of variables occurring in $\mathcal{L}_r$. Therefore, each state can be represented in a polynomial number of bits with respect to $|\varphi|$. Each tape of the automaton corresponds to a variable occurring in the set of linear constraints. For a certain input, the automaton checks whether the binary strings found on the tapes can be used as the binary representations of the corresponding variables in an assignment that satisfies the linear constraints. We assume that the representations of these variables are with leading 0s, so that their least significant bits are aligned. Intuitively, we encode a system of the form $A\vec{x} \leqslant \vec{b}^T$, where A is the coefficient matrix, $\vec{x}$ is the (column) vector of all variables, and $\vec{b}$ is the vector of integers. The state of the automaton computes $A\vec{y}(1..i)$, where $\vec{y}$ is a vector of integers whose binary representations are on the tapes of the automata, and $\vec{y}(1..i)$ is the vector containing each component, respectively, the integer whose representation consists of the most representative i bits from the representation of integer found at the corresponding position in $\vec{y}$. In order for $\vec{y}$ to be a solution, each component of the computed value $A\vec{y}(1..i)$ must stay between $-\ell$ and ℓ. Moreover, the transition from a state corresponding to $A\vec{y}(1..i)$ to the state corresponding to $A\vec{y}(1..i+1)$ can be computed from the current state and $\vec{y}(i+1)$. More detail on the construction of A is given in [54].

In our case, we additionally need to enforce that both the regular constraints on the binary representation of certain variables from $\mathcal{L}_b$, as well as the regular constraints on the other string variables that are not involved in any numstr predicate, are fulfilled.

Therefore, we augment the automaton described above in the following way: 1. We add a tape for each string variable $\mathtt{y} \in \mathcal{X}$ which does not occur in any string-number predicate, but appears in a linear length or regular membership constraint. From the point of view of the arithmetic part implemented by the automaton, these tapes are treated as if they represent variables which occur with coefficient 0 in the equations of the linear system we want to solve. In this context, as the letters on the respective tape are not involved in any arithmetic operation, we do not need to restrict the respective letters to the bits $\{ 0, 1 \}$. 2. We assume that the words on the tapes of the automaton have their last letters aligned. To this end, we can assume that our strings are padded with a prefix of special blank symbols, so that they all have the same length. The arithmetic part implemented by the automaton treats these blanks as 0s. The part of the automaton which checks the regular constraints simply neglects these blanks.

We will now explain how the automaton works. The arithmetic part was already described. The part checking the regular constraints works as follows. Suppose that the binary representation of the variable $\mathtt{x}$, which corresponds to the j^{th} tape of $\mathtt{A}$, must be in the language defined by a regular expression $R \in \mathtt{RegEx}_A$ (or, alternatively, not in the language defined by R). To this end, while $\mathtt{A}$ reads the representation of $\mathtt{x}$, as soon as we reach the first non-blank symbol on that tape, we also simulate the computation on $\mathtt{x}$ of the deterministic automaton corresponding to R or, respectively, to $\overline{R}$ (i.e., we construct the NFA M_R for R, and then simulate the transitions corresponding to R or $\overline{R}$ on the DFA obtained via the powerset construction from M_R as in the proof of Theorem 4.36). We accept the representations given as input to $\mathtt{A}$ if and only if $\mathtt{A}$ accepts them *and* they are also accepted by the automaton checking the components corresponding to variables that occur in $\mathcal{L}_b$. We simulate the states of the automata to enforce that the constraints from $\mathcal{L}_b$ have a polynomial number of components, and their total number is also polynomial.

The tapes corresponding to string variables which do not occur in string-number predicates are processed similarly. We simply simulate the computation of the deterministic automaton corresponding to the regular constraint on the string found on that tape.

To check whether there exists an input accepted by A in this way, we non-deterministically guess an input for A, by selecting one by one, from the most representative (left) to the least representative (right), the letters on the tapes of the automaton while storing at each step just the current guess, without saving the past guesses, and keep track of the current state of all the automata we simulate.

This process is clearly correct from the information we gave, and all the information we store fit in a polynomial number of bits. However, it is not clear that it terminates. For this, we show that we can bound the number of states of the automaton by a polynomial. Each of the automata corresponding to regular constraints, which we run when using A, has at most $P(|\varphi|)^{R(|\varphi|)}$ states for some polynomials P and R, and we run them in parallel on the tapes of A. The state corresponding to the linear system is a number between $-\ell$ and ℓ. We accept a guessed content of the tapes if and only if all automata accept it and the linear system is satisfied. Hence, we are essentially simulating a run of the product of these at most $Z(|\varphi|)$ automata, where Z is another polynomial. This product automaton accepts a non-empty language if and only if it accepts a word (sequence of columns of bits) whose length is at most its number of states. So, we must check whether it accepts a sequence of columns of bits of length $P(|\varphi|)^{Q(|\varphi|)Z(|\varphi|)}$. By keeping a binary counter, we can count how many guesses we have made, and stop (without having found a word) when we need to use more than $Q(|\varphi|)Z(|\varphi|)\log P(|\varphi|)$ bits for this counter.

If we can guess an assignment of the variables that satisfies $\mathcal{L}_b$ and $\mathcal{L}_l$ and the remaining regular constraints, then φ is satisfiable. If we cannot find any assignment, φ is not satisfiable.

Clearly, the decision procedure described works also in the case of the regular expressions containing complements. However, we cannot show the polynomial upper bound on the space we use. Therefore, $\mathcal{A}_{en}$ is decidable. □

While the general idea to prove the above result is based on a similar construction to that in Theorem 4.36, in this case we need to use a different strategy to work with the linear arithmetic constraints (due to the fact that string-number predicates are involved, and their fundamentally different nature with respect to the length function).

It is natural to ask whether the decidability result extends to the theories $\mathcal{A}_{enc}$ (and $\mathcal{A}_{snc}$), which also allow concatenation. While we leave this open, one can make two interesting observations. Firstly, these theories are expressive enough to define a predicate checking if two strings have equal length. Moreover, $\mathcal{A}_{enc}$ (and likewise $\mathcal{A}_{snc}$) has equivalent expressive power to the theory of word equations with regular constraints, a predicate allowing the comparison of the length of string terms, and the string-number predicate. The decidability of word equations with string-length comparisons is a long standing open problem, so we also consider it worthwhile to address the decidability of the slightly stronger theory $\mathcal{A}_{enc}$. According to [41], the theory of word equations, length constraints, and the string-number conversion is undecidable; the difference is that in that theory, one can check whether the length of a string term equals an integer term, which seems more general than what one can model in $\mathcal{A}_{enc}$. We get the following.

Theorem 4.41. *The satisfiability problem for $\mathcal{A}_{slnc}$ of regular expressions, linear integer arithmetic, a string-number predicate and concatenation is undecidable.*

Proof. We begin by looking at the theory $\mathcal{A}_{snc}$ and define a predicate $eqLen \subseteq \mathtt{Pat}_A \times \mathtt{Pat}_A$ defined by

$$eqLen(\alpha,\beta) \text{ iff } \mathsf{len}(\alpha) = \mathsf{len}(\beta)$$

for $\alpha,\beta \in \mathtt{Pat}_A$. We can express $eqLen(\alpha,\beta)$ as:

$$\begin{aligned} eqLen(\alpha,\beta) \;=\;\; & (\mathsf{z} \in 1\{0\}^*) \\ & \wedge \mathrm{numstr}(\mathsf{i},\mathsf{z}) \wedge \mathrm{numstr}(\mathsf{j},\mathsf{z}0) \wedge \mathrm{numstr}(\mathsf{n}_a,1\alpha) \\ & \quad \wedge \mathrm{numstr}(\mathsf{n}_b,1\beta) \\ & \wedge (\mathsf{i} \leqslant n_a) \wedge (\mathsf{n}_a + 1 \leqslant \mathsf{j}) \wedge (\mathsf{i} \leqslant \mathsf{n}_b) \wedge (\mathsf{n}_b + 1 \leqslant \mathsf{j}), \end{aligned}$$

for integer variables $\mathsf{i},\mathsf{j},\mathsf{n}_a,\mathsf{n}_b$ and string variables z. Indeed, for a potential assignment $h \in \mathcal{H}_{A\cup\mathbb{Z}}$, we have

$$h(\mathsf{i}) = 2^{len(\mathsf{z})} \text{ and } h(\mathsf{j}) = 2^{\mathsf{len}(\mathsf{z})+1}.$$

Then, we have

$$h(\mathsf{n}_a) = 2^{\mathsf{len}(\alpha)} + A \text{ and } h(\mathsf{n}_b) = 2^{\mathsf{len}(\beta)} + B,$$

where numstr(A, α) and numstr(B, β) are true. Therefore,

$$2^{\mathsf{len}(\mathsf{z})} \leqslant 2^{\mathsf{len}(\alpha)} + A < 2^{\mathsf{len}(\mathsf{z}+1)} \text{ and } 2^{\mathsf{len}(\mathsf{z})} \leqslant 2^{\mathsf{len}(\beta)} + B < 2^{\mathsf{len}(\mathsf{z})+1}.$$

It is immediate that $\mathsf{len}(\alpha) = \mathsf{len}(\beta) = \mathsf{len}(z)$, so our claim holds. We can also show that the theory of word equations with regular constraints and numstr predicate is equivalent to the theory $\mathcal{A}_{enc}$.

For one direction, we need to be able to express an equality predicate between string terms $eq \subseteq \mathtt{Pat}_A \times \mathtt{Pat}_A$. The regular constraints as well as those involving the numstr predicate are canonically encoded. This predicate is encoded as follows:

$$eq(\alpha, \beta) = eqLen(\alpha, \beta) \wedge \text{numstr}(\mathsf{i}, 1\alpha 1\beta) \wedge \text{numstr}(\mathsf{j}, 1\beta 1\alpha) \wedge (\mathsf{i} = \mathsf{j}),$$

for $\alpha, \beta \in \mathtt{Pat}_A$. Indeed, this tests for a potential assignment $h \in \mathcal{H}_{A\cup\mathbb{Z}}$ that

$$\mathsf{len}(\alpha) = \mathsf{len}(\beta) \text{ and } h(1\alpha 1\beta) = h(1\beta 1\alpha).$$

If these are true, it is immediate that $h(\alpha) = h(\beta)$.

For the converse, it is easy to see that each regular expression membership constraint $\alpha \,\dot{\in}\, R$ (respectively, $\alpha \,\dot{\notin}\, R$), where $\alpha \in \mathtt{Pat}_A$and $R \in \mathtt{RegExC}_A$, can be expressed as the word equation $\alpha \doteq \mathsf{x}_R$, where $\mathsf{x}_R \in \mathcal{X}$ is a fresh variable, which is constrained by the regular language defined by R (respectively, by the regular language defined by $\overline{R}$).

This allows us to define a stronger length-comparison predicate $leqLen \subseteq \mathtt{Pat}_A \times \mathtt{Pat}_A$, whose semantics are defined by

$$leqLen(\alpha, \beta) \text{ iff } \mathsf{len}(\alpha) \leqslant \mathsf{len}(\beta),$$

for $\alpha, \beta \in \mathtt{Pat}_A$. We can express $leqLen(\alpha, \beta)$ by

$$leqLen(\alpha, \beta) = (\mathsf{z} \in \{0, 1\}^*) \wedge eqLen(\alpha\mathsf{z}, \beta).$$

Finally, we can now move on to $\mathcal{A}_{elnc}$ and show our statement. According to [41] the quantifier-free theory of word equations expanded with numstr predicate and length function (not only a length-comparison predicate) and linear arithmetic is undecidable. Thus, if we consider $\mathcal{A}_{elnc}$, this undecidability result immediately holds according to the above.

□

In conclusion, $\mathcal{A}_{enc}$ and $\mathcal{A}_{eln}$ are the only fragments of $\mathcal{A}_{elnc}$ where the decidability status of the satisfiability problem remains open.

4.3.3 Design of an Algorithm Using Ideas of the Proof

As we have seen in the previous section, the complement with a regular expression does not play a significant role in practice. Therefore, we use the ideas within the decision procedure used in the proof of Lemma 4.34 to directly implement an algorithm to determine satisfiability of regular membership predicates together with length constraints.

Within the proof we construct a propositional logic abstraction of our input formula. This procedure is similar to the first step of a DPLL(T) procedure implemented within SMT-solvers. Formally we realise the approximation by using a mapping between atoms and propositional logic variables. In the following we specify this abstraction for arbitrary first order logic formulae.

Definition 4.42. Let $\mathcal{V}$ be a vocabulary. For a formula $\varphi \in \mathsf{FO}(\mathcal{V})$ such that $\texttt{bounded}(\varphi) = \varnothing$ let $\mathsf{v}_{\mathsf{PL}} : \texttt{atoms}(\varphi) \to \{ \mathsf{x}_i \mid \mathsf{x}_i \in \mathcal{X}, i \in [|\texttt{atoms}(\varphi)|] \}$ be a bijective mapping assigning a propositional logic variable to each atom of φ.

We define the propositional logic abstraction $\varphi_{\mathsf{PL}} \in \mathsf{PL}$ of φ inductively

$$\varphi_{\mathsf{PL}} = \begin{cases} \mathsf{v}_{\mathsf{PL}}(\varphi) & \text{if } \varphi \text{ is an atom,} \\ \neg\varphi'_{\mathsf{PL}} & \text{if } \varphi = \neg\varphi', \\ \varphi'_{\mathsf{PL}} \vee \varphi''_{\mathsf{PL}} & \text{if } \varphi = \varphi' \vee \varphi''. \end{cases}$$

Note, this definition fully specifies all quantifier-free propositional logic abstractions according to Remark 4.38 in Section 2.1.2. The key idea behind this abstraction is the fact that it allows us to identify whether an atom has to be satisfied by an assignment of the original propositional logic formula or not. Furthermore, if there not exists an assignment to the abstraction, the first order logic formula cannot be satisfiable, too.

Example 4.43. Consider the string constraint $C = \mathsf{x}_1 \dot{\in} 1^* \wedge \text{numstr}(15, \mathsf{x}_1) \wedge \mathsf{len}(\mathsf{x}_1) \geqslant 3$ where $\mathsf{x}_1 \in \mathcal{X}$ and $1 \in A$ seen in Example 4.33. We get the set

$$\mathtt{atoms}\,(C) = \{\, \mathsf{x}_1 \dot{\in} 1^*, \text{numstr}(15, \mathsf{x}_1), \mathsf{len}(\mathsf{x}_1) \geqslant 3 \,\}$$

of atoms. We might obtain the abstraction

$$\mathsf{v}_{\mathsf{PL}} = \{\, \mathsf{x}_1 \dot{\in} 1^* \mapsto \mathsf{y}_1, \text{numstr}(15, \mathsf{x}_1) \mapsto \mathsf{y}_2, \mathsf{len}(\mathsf{x}_1) \geqslant 3 \mapsto \mathsf{y}_3 \,\}\,.$$

for fresh variables $\mathsf{y}_1, \mathsf{y}_2, \mathsf{y}_3 \in \mathcal{X}$. All together we obtain the propositional logic abstraction

$$C_{\mathsf{PL}} = \mathsf{y}_1 \wedge \mathsf{y}_2 \wedge \mathsf{y}_3.$$

We outline the idea of this procedure including the split of the atoms into separate sets as seen in the proof in Algorithm 3.

Algorithm 3: Procedure to split an input formula into linear and regular membership constraints with respect to their satisfiability.

```
Input  : Formula $\varphi$ in conjunctive normal form over theory $\mathcal{A}_e$
Output : Sets $L_R$ and $L_L$ of regular membership predicates and linear length constraints
$\varphi_{\mathsf{PL}}$ = calculateBooleanAbstraction($\varphi$);
if not checkSAT($\varphi_{\mathsf{PL}}$) then
    return $\varnothing, \varnothing$
$b$ = getAssignment($\varphi_{\mathsf{PL}}$);
$L_R, L_L := \varnothing, \varnothing$;
forall $\alpha \dot{\in} R \in$ atoms ($\varphi$) do
    if $b(\mathsf{v}_{\mathsf{PL}}(\alpha \dot{\in} R))$ then
        $L_R = L_R \cup \{\, \alpha \dot{\in} R \,\}$
    else
        $L_R = L_R \cup \{\, \alpha \dot{\in} \overline{R} \,\}$
forall $\sum_{\mathsf{x} \in \mathcal{X}} c_{\mathsf{x}} \cdot \mathsf{x} \bowtie c \in$ atoms ($\varphi$) do
    if $b(\mathsf{v}_{\mathsf{PL}}(\sum_{\mathsf{x} \in \mathcal{X}} c_{\mathsf{x}} \cdot \mathsf{x} \bowtie c))$ then
        $L_L = L_L \cup \{\, \sum_{\mathsf{x} \in \mathcal{X}} c_{\mathsf{x}} \cdot \mathsf{x} \bowtie c \,\}$
    else
        $L_L = L_L \cup \{\, \sum_{\mathsf{x} \in \mathcal{X}} c_{\mathsf{x}} \cdot \mathsf{x} \not\bowtie c \,\}$
return $L_R, L_L$
```

Solving regular expression membership predicates in practice is an expensive task. The natural way of constructing a deterministic finite automaton based on a regular expression to ease the satisfiability suffers not only space but also time. To this extend we tried to postpone the construction of a deterministic finite automaton as much as possible. To do so we enriched the above mentioned decision procedure by a refinement step, mostly to quickly determine unsatisfiability of an input formula, but also to leverage information we usually obtain by expensive automata constructions.

The idea is similar to the idea we use in Section 4.1.2 in Definition 4.11 for word equations: we construct a length abstraction based on a regular expression membership predicate. This abstraction allows avoiding the construction of the unary automata we use in the proof to obtain the length constraints via Chrobak normal form [34]. We will first show how to get a length approximation of a regular expression.

Definition 4.44. Let $R \in \texttt{RegExC}_A$ be a regular expression. We define the *length abstraction of a regular expression* recursively by

$$\mathsf{rAbs}(R) = \begin{cases} \{1\} & \text{if } R = a \in A, \\ \{0\} & \text{if } R = \varepsilon \\ \varnothing & \text{if } R = \varnothing, \\ \mathsf{rAbs}(R_1) \cup \mathsf{rAbs}(R_2) & \text{if } R = R_1 \cup R_2, \\ \{ n + m \mid n \in \mathsf{rAbs}(R_1), m \in \mathsf{rAbs}(R_2) \} & \text{if } R = R_1 \cdot R_2, \\ \{ \ell \cdot n \mid \ell \in \mathbb{N}_{>0}, n \in \mathsf{rAbs}(R_1) \} \cup \{0\} & \text{if } R = R_1^*, \\ \mathbb{N} \backslash \mathsf{rAbs}(R_1) & \text{if } R = \overline{R_1}. \end{cases}$$

for $R_1, R_2 \in \texttt{RegExC}_A$.

We naturally combine Definition 4.11 and the abstraction for regular expressions as follows.

Definition 4.45. Let $\alpha \dot{\in} R$ be a regular membership predicate for $\alpha \in \texttt{Pat}_A$ and $R \in \texttt{RegExC}_A$. Furthermore let $\alpha_\# = \sum_{\mathsf{x} \in \mathcal{X}} |\alpha|_{\mathsf{x}} \cdot \bar{\mathsf{x}} + \sum_{a \in A} |\alpha|_a$ be the length abstraction of α. We call the system of linear equalities

$$\{ \alpha_\# - r = 0 \mid r \in \mathsf{rAbs}(R) \}$$

the *length abstraction* of $\alpha \dot{\in} R$.

Whenever $\not\models \{ \alpha_\# - r = 0 \mid r \in \mathsf{rAbs}(R) \}$ for a regular membership predicate $\alpha \dot{\in} R$, we immediately get the unsatisfiability of $\alpha \dot{\in} R$, since the corresponding regular expression R does not represent a suitable length for our pattern α. It is worth mentioning that whenever we want to solve a negated regular membership predicate $\alpha \mathrel{\dot{\notin}} R$ we use the same method as presented above but solve $\alpha \mathrel{\dot{\in}} \overline{R}$.

In practice, whenever we encounter a Kleene star within a regular expression, we introduce a fresh integer variable. It is worth mentioning that nested stars do not break the requirement of forming a linear equality. We simply reuse a previous variable. The following example gives an intuition.

Example 4.46. Consider the regular membership predicate

$$\mathsf{x}_1 ab \mathsf{x}_2 \mathrel{\dot{\in}} a(a^* \cup b)^*.$$

The abstraction of the pattern $\mathsf{x}_1 ab \mathsf{x}_2$ is given by $\bar{\mathsf{x}}_1 + \bar{\mathsf{x}}_2 + 2$. The abstraction of the regular expression is constructed as follows:

$$\begin{aligned}
&\mathsf{rAbs}\left(a(a^* \cup b)^*\right) \\
&= \{\, 1 + m \mid m \in \{\, \ell \cdot n \mid n \in \{\, 1 \cdot \ell' \mid \ell' \in \mathbb{N}_0 \,\} \cup \{\, 0 \,\}, \ell \in \mathbb{N}_0 \,\} \\
&\qquad\qquad\qquad \cup \{\, 1 \,\} \cup \{\, 0 \,\} \,\} \\
&= \{\, 1 + m \mid m \in \{\, \ell \cdot \ell' \mid \ell, \ell' \in \mathbb{N}_0 \,\} \cup \{\, 0 \,\} \,\} \\
&= \{\, 1 + m \mid m \in \mathbb{N} \,\}
\end{aligned}$$

It is easy to see, that is is sufficient to introduce a single variable $\bar{\mathsf{y}}_1$ instead of introducing a single one for each Kleene-star occurring within the regular expression. In total we obtain the length abstraction

$$\bar{\mathsf{x}}_1 + \bar{\mathsf{x}}_2 + 2 = 1 + \bar{\mathsf{y}}_1$$

describing all suitable length for our solutions to the initial regular membership predicate.

In Algorithm 4 we outline the idea of using the length abstraction. Note, that this algorithm is not necessary to form a decision procedure for formulae over $\mathcal{A}_{elc}$ but delivers a valuable addition, allowing to quickly check whether a formula is unsatisfiable.

Continuing with the ideas seen in the proof of Lemma 4.34, whenever the string constraint C contains multiple regular expression membership constraints asking whether the same pattern $\alpha \in \mathtt{Pat}_A$ belongs to multiple regular languages indicated by regular expressions $R_1, \ldots, R_k \in \mathtt{RegExC}_A$ for $k \in \mathbb{N}$, meaning $\alpha \mathrel{\dot{\in}} R_1, \ldots, \alpha \mathrel{\dot{\in}} R_k$, we have to ensure all constrains

Algorithm 4: Procedure to determine unsatifiablity of a set of regular membership predicates and linear length constraints.

```
Input   : Sets $L_R$ and $L_L$ of regular membership predicates and linear length constraints
Output  : UNSAT or UNKNOWN
$R_{\mathsf{abs}}$ = calculateLengthAbstraction($L_R$);
if not checkSAT($\bigwedge_{l \in R_{\mathsf{abs}} \cup L_L} l$) then
    return UNSAT
return UNKNOWN
```

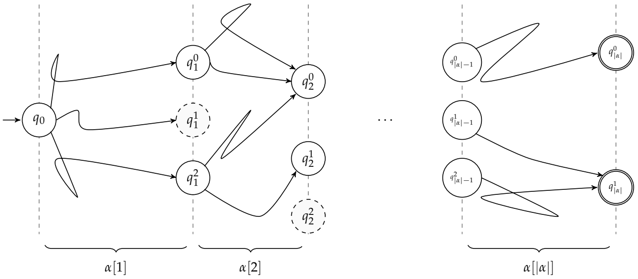


Figure 4.9. Selecting initial and accepting state for all letters and variables within our pattern α.

simultaneously. As seen in the proof, we construct non-deterministic automata A_i such that $L(A_i) = L(R_i)$ for all $i \in [k]$ and simply combine them by using the product automaton construction (cf. [104]) to construct an automaton A such that $L(A) = \bigcap_{i \in [k]} L(A_i)$. We construct this automaton lazily on the fly, which potentially also reveals useful information about our constraints: whenever no accepting state in the product automaton A is reachable, $\alpha \mathrel{\dot\in} R_1, \ldots, \alpha \mathrel{\dot\in} R_k$ cannot be fulfilled at the same time. If the intersection is not empty we constructed the whole automaton A which we will reuse later within our decision procedure. To this extend we will now focus on solving $\alpha \in L(A)$ instead of solving $\alpha \mathrel{\dot\in} R_1, \ldots, \alpha \mathrel{\dot\in} R_k$. The general idea of this algorithm is outlined in Algorithm 5.

The next step is non-deterministically guessing starting states for each letter or variable $\alpha[i]$ for $i \in [|\alpha|]$ within each pattern $\alpha \in \mathtt{Pat}_A$ of question in the reformulated regular membership predicate $\alpha \in L(A)$. Our algorithm determines this step by trying all different combinations of states such that all together they form a valid path from the initial state to an accepting

Algorithm 5: Procedure for constructing the intersection automaton for a set of regular membership constraints

```
Input  : Set $L_R$ containing regular membership predicates over theory $\mathcal{A}_e$
Output : A set $L_{\mathsf{Aut}}$ containing tuples of pattern and product automaton for all equal left hand sides
$L_{\mathsf{Aut}} = \emptyset$
forall $\alpha \in \mathtt{Pat}_A$ such that there exists $R \in \mathtt{RegExC}_A$ and $\alpha \dot{\in} R \in \mathtt{atoms}\,(\varphi)$ do
    forall $\alpha \dot{\in} R_i \in \mathtt{atoms}\,(\varphi)$ do
        $A_i = \mathtt{calculateFiniteAutomaton}(R_i)$
    $A = \mathtt{calculateProductAutomaton}(R_1, \ldots, R_k)$;
    if $L(A) = \emptyset$ then
        /* Empty intersection, formula unsatisfiable */
        return $\emptyset$
    $L_{\mathsf{Aut}} = L_{\mathsf{Aut}} \cup (\alpha, A)$;
return $L_{\mathsf{Aut}}$
```

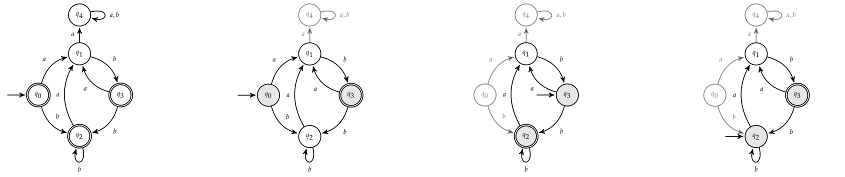
state through the automaton A. Therefore, we obtain several automata, each of them corresponds to a single possible solution $w_i \in A^*$ such that $h(\alpha[i]) = w_i$. Whenever, a variable is occurring multiple times within a pattern α, we intersect the resulting automata for each variable occurrence i in α. Thus, if the intersection is non empty, we obtain an automaton again describing all possible solutions for the considered variable within the current split which fulfils the regular expression membership predicate. We visualise this idea in Figure 4.9, where for each letter or variable $\alpha[i]$ we pick a state q_i and check whether there is a path to the previous state q_{i-1}. Not all sub-automata lead to an accepting state as visualised with the dashed states. Moreover, if we are not able to construct such automata, the initial formula does not have a solution.

Naturally, whenever more than one regular membership constraint is involved we have to check whether all of them accept at least a single common word. To this extend we intersect all resulting automata describing a potential solution to a variable. Once again, if the product automaton does not accept a single word, the initial formula does not have a solution.

The idea of this procedure is outlined in Algorithm 6. Line 2-21 construct all possible automata for each variable, while line 24-45 check whether there is a common solution. The following example deepens the idea.

Example 4.47. Consider the regular membership constraint

$$\alpha = \mathsf{x}_1\mathsf{x}_2\mathsf{x}_1 \dot{\in} (\varepsilon \cup bb^*) \cup (a \cup bb^*a)b((a \cup bb^*a)b)^*(\varepsilon \cup bb^*) = R.$$



a. initial automaton

b. automaton describing solutions for the first occurrence of x_1

c. automaton describing solutions for x_2

d. automaton describing solutions for the second occurrence of x_1

Figure 4.10. Automata accepting the same language as the regular expression $(\varepsilon \cup bb^*) \cup (a \cup bb^*a)b((a \cup bb^*a)b)^*(\varepsilon \cup bb^*)$ and example splits.

In Figure 4.10 a we visualise the automaton A such that $L(R) = L(A)$. We are assuming our algorithm selected the sequence q_0, q_3, q_2, q_3 as potential splits for our pattern α. Therefore, we have to construct automata such that there exists sub-automata for x_1 having the initial state q_0 and accepting state q_3, for x_2 having the initial state q_3 and accepting state q_2, and for the second occurrence of x_1 an automaton having the initial state q_2 and accepting state q_3. Luckily, all chosen accepting and initial states are connected within our original automaton A. Therefore, we extract the sub-automata visualised in Figure 4.10 b - d describing all potential solutions to our variables. Selected states are highlighted in light grey and unreachable states, respectively states not having a path to the accepting state are faded. Since x_1 is occurring more than once in our pattern α, we have to intersect all automata corresponding to x_1. The result is exactly the automaton depicted in Figure 4.10 b. Now we can pick two arbitrary words being accepted by our automata to obtain a solution to our regular expression membership predicate $\alpha \mathrel{\dot\in} R$. We might pick $h(\mathsf{x}_1) = bab$ and $h(\mathsf{x}_2) = ab$, which is a suitable assignment. Therefore, $h \models \alpha \mathrel{\dot\in} R$.

Whenever we find a valid set of potential solutions, we again use the length abstraction seen in Definition 4.45. As also seen in the proof, this is sufficient, since these abstractions now directly correspond to a potential solution for each variable. We simply use an integer solver and ask whether at least one of the abstractions of our automata combined with the linear length constraints stated within our original formula φ is satisfiable. We outline this idea in Algorithm 7.

In order to obtain an assignment for all variables, we can simply use

Algorithm 6: Procedure to calculate potential solutions to a set of regular membership constraints

```
Input  : Set $L_R$ containing regular membership predicates over theory $\mathcal{A}_e$
Output : A set $L_{\mathsf{Aut}}$ containing tuples of patterns and product automaton for all equal left hand sides
$sols, symbols = \varnothing, \varnothing$
forall $(\alpha, A) \in L_{\mathsf{Aut}}$ do
    Let $A = (Q, A, \delta, q_0, F)$;
    $n = |\alpha|$;
    $M = \varnothing$;
    $symbols = symbols \cup \{ \alpha[i] \mid i \in [|\alpha|] \}$
    forall $q_1, q_2, \dots, q_{n-1} \in Q^{n-1}$ do
        forall $q_n \in F$ do
            $C = \{ \alpha[i] \mapsto \varnothing \mid i \in [n] \}$ forall $i \in [n]$ do
                if $\neg\exists\, w \in (A \cup \mathcal{X})^{*} . \delta(q_{i-1}, w) = q_i$ then
                    break and select new states (if any left)
                $A_{\alpha[i]} = (Q, A, \delta, q_{i-1}, \{ q_i \})$
                if $C(\alpha[i]) \neq \varnothing$ then
                    $I_{\alpha[i]} = \mathtt{calculateProductAutomaton}(A_{\alpha[i]}, C(\alpha[i]))$
                    if $L(I_{\alpha_i}) = \varnothing$ then
                        break and select new states (if any left)
                    else
                        $C(\alpha[i]) = I_{\alpha[i]}$
                else
                    $C(\alpha[i]) = A_{\alpha[i]}$
        $M = M \cup \{ C \}$
    if $M = \varnothing$ then
        /* we have not found a single matching split */
        return $\varnothing$
    if $sols \neq \varnothing$ then
        $C = \varnothing$
        forall $S \in sols$ do
            forall $S' \in M$ do
                $CC = \varnothing$
                forall $x \in symbols$ do
                    if $x \in \mathrm{dom}(S)$ then
                        $I_x = \mathtt{calculateProductAutomaton}(S(x), S'(x))$
                        if $L(I_x) = \varnothing$ then
                            $CC = \varnothing$;
                            break and select a new set $S'$ (if any)
                        else
                            $CC(x) = I_x$
                    else
                        $CC(x) = S(x)$
                $C = C \cup \{ CC \}$
        if $C \neq \varnothing$ then
            $sols = C$
        else
            /* the intersection of the solutions is empty; formula unsatisfiable */
            return $\varnothing$
    else
        $sols = M$
return $sols$
```

Algorithm 7: Procedure to check the validity of calculated solutions with respect to given linear constraints

```
Input   : Set $sols$ of automata describing potential solutions to our initial formula and a set of linear length constraints $L_L$
Output  : SAT or UNSAT of the original formula
1 forall $S \in sols$ do
2     if checkSAT($\bigcup_{x \in \mathsf{dom}(S)}$ calculateLengthAbstraction($x \in S(x)$) $\cup L_L$) then
3         return $SAT$
4 return $UNSAT$
```

the constructed automata for each variable and take an arbitrary accepted word. Note, we naturally have to use the automata where our procedure in Algorithm 7 reached line 3 and therefore determined satisfiability of the input formula.

We now presented a couple of algorithms which all together form a decision procedure for $\mathcal{A}_{elc}$. We simply connect Algorithm 3 and Algorithm 5-7. The soundness of the procedure follows directly from the proof of Lemma 4.34. Termination is also guaranteed, since we are always iterating over finite sets.

4.3.4 Related Work

Closely related to our work is the decidability result of $\mathcal{A}_{elc}$ in [4]. Unfortunately, no proof is given within their publication. From their description we deduce the choice of another proof strategy. Other strongly related complexity results to ours are presented in [83, 84]. Furthermore, [41] investigates sub-theories involving regular membership constraints. Contrary to our work, none of the above mentioned publications mentions a directly derived implementation of their proof ideas. On the other end, many modern solvers typically handle regular expression membership constraints via an automata-based approach (cf. [11]). Automata-based methods are powerful and intuitive, but solvers must handle two key practical challenges in this setting, which we also highlighted earlier: the first challenge is that many automata operations, such as intersection, are computationally expensive, yet handling these operations is required in order to solve constraints that are relevant to real-world applications. The second challenge relates to the integration of length information with regular expression membership constraints. Length constraints derived from automata may imply a disjunction of linear constraints, which is often more challenging for solvers to handle than a conjunction. Both of these challenges where efficiently handled as we showcase in Section 6.4 which again highlights the novelty of our approach.

4.3.5 Conclusion

The analysis of several string solving benchmarks containing regular expression membership queries revealed relevant sub-theories based around regular membership predicates. It turned out that the most frequently occurring sub-theory is actually decidable. Next to proving the PSPACE-completeness of this theory, we used the ideas of these proofs directly to develop an SMT-solver for regular expression membership predicates which outperforms several state of the art string solver (see Section 6.4). Therefore, we show that an interleaving between theory and practice potentially leads to new interesting solutions in both worlds. Our future work will continue on this trail to obtaining relevant sub-theories used in practice, always in the hope of finding decidable sub-theories which lead to the design of new decision procedures for solving practically relevant string constraints.

Chapter 5

Ensuring the Correctness of String Solvers

"In einer Freundschaft wie dieser gibt es kein Zurück."

TOCOTRONIC

During the development of the techniques presented in Chapter 4 we realised that it was relatively hard to test our implementation against existing solvers due to a few reasons:

- there was no general collection of standard string solving benchmarks: it seemed very hard to identify the relevant benchmarks for string solving tasks, without considering a high amount of literature, and collecting from each paper the input data they used to test their tools;
- the benchmarks we found in the way described above were largely uncategorised as to which kind of string constraints they contained;
- benchmarks from different tools use different input formats or slight variations of the same core format (SMT-LIB);
- the satisfiability/unsatisfiability verdicts provided by the tools used in the literature were wrong on many benchmarks; and, finally,
- benchmarks were huge and it is really hard to get an overview where one solver was better than the other.

Naturally, this is a result of a field growing organically and a lack of standardisation. Luckily, efforts gradually have converged and, for instance, currently we have a unified string logic standard as part of SMT-LIB. That is however only one step towards easily comparing string solvers, addressing, at least partly, the second and third item from the above list.

To help this effort of making the testing and evaluation of string solvers easier and standardised, we have developed ZaligVinder. At its core, ZaligVinder has a collection of benchmarks, extracted from the literature related to many different tools. It allows running all of the established string solvers on the benchmark-sets and it includes a graphical overview of the results. Also, to overcome the problem that the existing benchmarks verdict is wrong ZaligVinder includes a cross-validation mechanism for asserting what is the correct result for inputs with unknown verdicts. We use this mechanism to annotate the benchmarks with reliable results. While doing so we apply mechanisms to automatically ease human readability. Our approach, thus, addresses the first, second, fourth, and fifth items from our list.

In this chapter we start by explaining our benchmarking framework ZaligVinder. We give an introduction on how to perform an analyse using ZaligVinder, how to analysis the resulting data with our tool by explaining the different techniques. Afterwards, we summarise the set of collected benchmarks which consist of a total of 114,475 instances split over 152 different benchmark sets. Unfortunately, these benchmarks are often annotated with false results. In the next Section we extract a sophisticated set of 112,831 benchmarks, annotated, this time, with reliable results: the SAT-instances are provably correct, while the UNSAT-instances are labelled according to the result reported by the majority of the state-of-the-art solvers. We note that solving formulae involving strings usually consists (although not always explicitly stated) in executing several algorithms in sequence. In particular, it seems that a good preprocessor, which simplifies and restructures the input data in a more concise way, often leads to a better performance of a string solver. Thus – as a proof of concept – we extended the tool our tool to cope with swappable preprocessing techniques – not only to use it for performance tests of string solvers, but also to export the resulting mutated instances. This extension is presented in Section 5.1.2. In Section 5.3 we perform two case studies on how to use ZaligVinder for a specific analysis of a string solver and how to extend the tool to use the resulting data for further post processing steps.

Before concluding, we evaluate our framework based on several research questions and address related work.

This chapter ist based on [76] published at AST 2020 and on [77] published in the Journal of Software: Evolution and Process 2021.

5.1 ZaligVinder

One of the main focal points is collecting existing benchmarks and setting up a framework for easy comparison of string solvers. In this section we show how to setup our framework ZaligVinder with the tools (CVC4 [16]and Z3str3 [23]), and the collected benchmarks. We also discuss how to extend the number of tools and the benchmark-sets. Finally we show how to start the graphical comparison interface of ZaligVinder and review individual components of our framework.

5.1.1 Performing an analysis

After downloading ZaligVinder from

`https://github.com/zaligvinder/zaligvinder`

the first thing one has to do is setup a benchmark-setup file. This is a Python 3 script that sets up which benchmark-sets constitutes the current test setup, selects which solvers to execute on, and a common timeout for each tool.

```python
import utils
import storage
import voting.majority as voting
from runners.base import TheRunner as testrunner
import summarygenerators
import startwebserver

import tools.z3str3
import tools.cvc4
import tools.z3seq
import models.woorpje
```

```
12
13 timeout = 30
14 ploc = utils.JSONProgramConfig ()
15 voter = voting.MajorityVoter()
16 verifiers = ["cvc4","z3seq"]
17
18 store = storage.SQLiteDB ("Example")
19
20 summaries = [ summarygenerators.terminalResult,
       ↪
21               store.postTrackUpdate]
22
23 tracks = models.woorpje.getTrackData ("Woorpje
       ↪ Word Equations")
24
25 solvers = {}
26 for s in [tools.z3str3,tools.cvc4]:
27     s.addRunner (solvers)
28
29 testrunner().runTestSetup (tracks,solvers,voter
       ↪ ,summaries,store,timeout,ploc,verifiers)
       ↪
30 startwebserver.Server (store.getDB ()).
       ↪ startServer ()
```

Listing 5.1. Basic setup script

In Listing 5.1 we show a basic benchmark-setup file. The file starts with `import`ing helper modules for storage, and a voting mechanism for deducing reference results for input models which we do not know the correct categorisation of ("satisfied" or "not satisfied") for.

The Kaluza benchmark-set 5.2 is known to be an example of wrongly provided answers. In lines 8 to 11 we import the Z3str3, Z3seq and CVC4 driver module and a collection of models called `woorpje`. In line 13 through line 16 we set the timeout, pick a configuration manager, a voting mechanism including the verification procedure and specify the validating solvers. The configuration manager (`utils.JSONProgramConfig`) object is used to locate binaries on the host machine. The storage mechanism (an SQLite database) is setup in line 18 and in line 20 we setup a number of functions that will be run after each track. In particular, we will generate an output to the terminal and perform a `postTrackUpdate` to the storage. The later one updates the results according to the validation result. In line 25 through line 27 we pick up all the tracks (lists of input files) from the `models.woorpje` module and collects them under the common name `Woorpje Word Equations` and put the Z3str3 and CVC4 solvers into the collection of solvers that will be executed. In line 29 we execute this entire benchmark. Finally, after having executed the benchmarks a webserver is started where the results can be inspected.

Adding Benchmark Input Files

In ZaligVinder we represent an input file to the tools as a `TrackInstance` object which consist of a name, a path to the input file and an expected result (`True` for satisfiable, `False` for not satisfiable and `None` for unknown). The `TrackInstance` objects are gathered into `Track` objects having a name, the `TrackInstance` objects and a benchmark name. This benchmark name is useful for grouping several tracks under a common name for presentation purposes.

To make new input files available to ZaligVinder, a Python submodule should be added, namely the models module of ZaligVinder. The initialisation file (__init__.py) of the sub-module should contain a function `getTrackData` accepting a single parameter being the name we want to group the tracks under. A prototypical implementation of such an __init__.py can be seen in Listing 5.2. It simply iterates over all files in the directory, and if the filename suggests it being a SMT instance then a track instance is made. Finally, a list containing only a single `Track` is returned.

```
import os
import utils
dir_path = os.path.dirname(os.path.realpath(
    ↪ __file__))

def getTrackData (bname = ""):
    filest = []
    for root, dirs, files in os.walk(dir_path,
        ↪ topdown=False):
        for name in files:
            if name.endswith (".smt"):
               filest.append(utils.
                   ↪ TrackInstance(name,os.
                   ↪ path.join(root,name)))

    return [utils.Track("Track 1",filest,bname)
        ↪ ]
```

Listing 5.2. Prototypical file to add input files to ZaligVinder

Adding a Solver

In ZaligVinder the driver interface is encapsulated into sub-modules of the tools module. The driver module must expose one function `addRunner` accepting a `dictionary` and add a runnable under a solver specific name. The runnable added to this dictionary must accept four parameters `eq`, `timeout`,`ploc`,`wd` where `eq` is the path to an input file, `timeout` is the user specified timeout, `ploc` is the configuration manager from Listing 5.1 and `wd` is a directory the solver can use for temporary storage. Here we will not go into details about how each individual solver is run, but show a standard structure for the solver file in Listing 5.3. What should be mentioned is, that the runnable (`run` function) returns a `utils.Result` object which encapsulates the verification result, the time it took, whether the solver timed out, the standard output stream from the tool and the generated model of the tool.

```
import subprocess
import tempfile
import os
import utils
import sys
import timer

def run (eq,timeout,ploc,wd):
    path = ploc.findProgram ("Solvername")

    time = timer.Timer ()
    try:
        out = subprocess.check_output ([path,eq
            ↪ ],timeout=timeout).decode().
            ↪ strip()
    except subprocess.TimeoutExpired:
        return utils.Result(None,timeout,True
            ↪ ,1)
    except subprocess.CalledProcessError:
        return utils.Result(None,timeout,False
            ↪ ,1)

    time.stop()

    if "unsat" in out:
        return utils.Result(False,time.getTime
            ↪ (),False,1,out)
    elif "sat" in out:
        return utils.Result(True,time.getTime()
            ↪ ,False,1,out,"\n"
                                .join(out.split
                                    ↪ ("\n")
                                    ↪ [1:]))
    return utils.Result(None,time.getTime (),
        ↪ False,1,out)

def addRunner (addto):
    addto['Solvername'] = run
```

Listing 5.3. Prototypical Solver file

Using multiple cores

In Listing 5.1, line 4 we load the runner for handling the execution of a particular solver. In this example we load our base runner which holds a function to subsequently execute a solver on all instances of an asked track. To shorten the evaluation time ZaligVinder offers a multi core setup being based on Pythons `multiprocessing` package. Switching to our multi core setup is made as easy as possible. We simply load the multi runner by replacing line 4 by `from runners.multi import TheRunner as testrunner`. `TheRunner` which now takes an optional integer which corresponds to the amount of cores being used during the analysis. ZaligVinder has now successfully been configured for a multi core run.

Verification

Many of our gathered benchmark sets do not contain expected associated results. As written above, some of them even contain incorrect classifications. In order to find misclassified instances by a solver ZaligVinder offers two different techniques: 1. model validation 2. result voting. Whenever a solver classifies an instances as satisfiable the string solving guidelines expect the solver to return a valid model. To check validity of a model the corresponding variables are substituted into the input formula. The resulting variable free formula is then asserted and a set of solvers – seen in Listing 5.1, line 16 – verifies its satisfiability. If all verification solvers treat a model as valid, ZaligVinder marks the instances as correctly solved. Since string solvers are not required to produce a proof for non satisfiable cases, we can not use a similar procedure as described above. Therefore, ZaligVinder performs a majority voting considering all results returned for an instance. If at least one solver returned a valid model, being verified by the above procedure, we treat an instance as satisfiable. A majority voting is performed between instances being classified as unsatisfiable or unknow, respectively, timeout, we treat an instances as unsatisfiable if more solvers were able to classify an instances as unsatisfiable. However, an expected result is only set whenever the above conditions are met.

5.1.2 Further analysis mechanisms

Most state of the art string solvers support an extensive set of high-level string operations. These operations can often be reduced to simpler forms only with a few different operations (cf. [97]). Consider, e.g., the equality $u \doteq \texttt{replace}(v, u, v)$ for arbitrary string terms u and v, as implemented within CVC4; clearly, it is sufficient to solve $u \doteq v$ due to the semantics of `replace`. There exist a multitude of such rules, which drastically simplify and clarify the input data, and are usually applied before or during the actual run. Generally, these reductions can be seen as a separate algorithm, and – in principle – should be interchangeable between different solvers.

Our goal is to allow users applying a sequence of different algorithms to test the abilities of their own string solver.

Within this section we describe an initial step, by presenting a mechanism allowing us to use different preprocessing algorithms.

Swappable preprocessing

We introduce an extension for ZaligVinder which allows applying preprocessing algorithms before the string solvers starts the search for a solution. We implemented a new class `Preprocessor` which is invoked within the solver setup seen in Listing 5.3 within the `run` function as outlined in Listing 5.4.

```
1 from preprocessor.preprocessor import *
2 def run (eq,to,ploc,wd,store_instances=False):
3     ...
4     tempd = tempfile.mkdtemp ()
5     ppfile = os.path.join (tempd,"pp.smt")
6     p = Preprocessor(ploc,"solver",["add","pars
          ↪ "])
7     p(eq,ppfile,store_instances)
8     time = timer.Timer ()
9     ...
```

Listing 5.4. Prototypical Solver file including an external preprocessor

Next to `import`ing the preprocessing class in line 1 all that is needed is creating a temporary directory (line 5), instantiating the preprocessor (line 6), and calling that instance (line 7). The instance requires three parameters, the first two being the program location setup `ploc` which is simply looking up the name given as second parameter within the program configuration of ZaligVinder. The third parameter holds a list of additional parameters. These parameters might be needed to only trigger a specific algorithm of the external solver. We expect the preprocessing procedure to output its results as an SMT-LIB instance which will directly be used within the actual solver call. The call to the preprocessing technique gets the instance path, an output path, and an optional parameter indicating whether the resulting preprocessed instance should be stored in the file system, creating a new set of mutated testing instances.

The preprocessing class itself performs the following steps:

1. The input instances are cleaned up, since we cannot always expect an instance to only contain SMT-LIB operators a particular solver is able to handle. This step removes all comments, sets the SMT-logic to the most general one (`ALL`), and removes some SMT-LIB operations, e.g., asking to quit the search early, etc.. Afterwards we translate all operations and escape sequences not following the SMT-LIB 2.5 syntax into appropriate ones.
2. The resulting formula is then passed to an external preprocessing mechanism, which is expected to return an SMT-LIB-formula as well.
3. Lastly, a postprocessing-step behaving similar to the preprocessing step is executed. This procedure cleans the resulting output file from operations causing undesired behaviour.

For example, consider $\mathsf{x}_1 \doteq \texttt{str.replace}(\mathsf{x}_2,\mathsf{x}_1,\mathsf{x}_2) \wedge \mathsf{x}_1 \doteq aba$ where, again $\mathsf{x}_1,\mathsf{x}_2$ are variables and a, b are constants, then applying the aforementioned simplification rule leads to solving a much simpler string constraint $\mathsf{x}_1 \doteq \mathsf{x}_2 \wedge \mathsf{x}_1 \doteq aba$.

5.1.3 Post-Processing

ZaligVinder stores the benchmarking results into an SQLite database as this enables easy post-processing of the data. Next to a REST API offering relevant data in JSON format we have implemented a basic graphical user interface (GUI) using this database. Thirdly, we provide a collection of `Python3` scripts to generate LaTeX plots and tables as well as a Markdown website to review and share the results of a run. The GUI and the API are started by running `python startwebserver.py dbfile`, where `dbfile` is the database file created by ZaligVinder and guide a web browser to `http://localhost:8081`[1].

Database

The database model was designed to ease the post processing abilities. In Figure 5.1 we depict our database as an entity relationship diagram.

[1] A demonstration and further explanations are available at `http://zaligvinder.github.io`

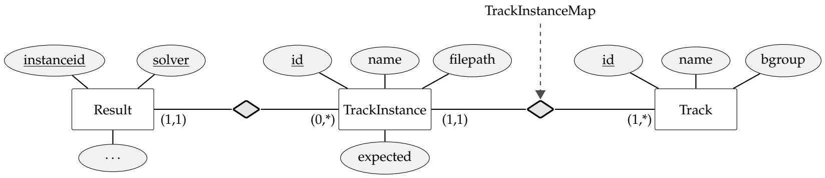


Figure 5.1. Entity relationship diagram of ZaligVinder's database schema. For ease of readability the `Result` entity omits the attributes `timeouted`, `result`, `time`, `output`, and `model`.

The database holds the four tables `Result`, `TrackInstance`, `TrackInstanceMap`, and `Track`, where `TrackInstanceMap` forms the relationship between `Track` and `Instances`. The `Result`s entity holds a specific result of a solver on a particular instance. We store the running times, the solver's returned result, a flag, whether the solver timeouted, the solver's output on this particular instances, the model – in case of determining the instance as satisfiable, and a verified flag. The result is threefold: it can either be `SAT`, `UNSAT`, or `UNKNOW`. Due to undecidabilty of certain string solving fragments, all string solvers implement incomplete algorithms. Therefore, the result `UNKNOW` requires further investigation: it either indicates an early give up to finding a solution or a timeout. The later fact can be observed by considering the `timeouted` flag which is only true whenever a solver got an external kill signal. The verified flag indicates whether a return model is valid. However, for an instance which is declared unsatisfiable we fall back to the majority voting as described in 5.1.1.

The `TrackInstance` entity holds all registered formulae being identifiable by an `id`. The instance holds a name – usually the file name of the input formula, a location path within the file system, and the expected result of the instance. If this result is not known prior to the run we use the same procedure as described in 5.1.1 to fill the expected result. Thus, it cannot be fully trusted.

The `Track` entity consists of track names and their corresponding benchmark groups. The instances of each track are linked via the relation `TrackInstanceMap` using the corresponding primary keys.

Table 5.1. Rest API entry points and their functionalities

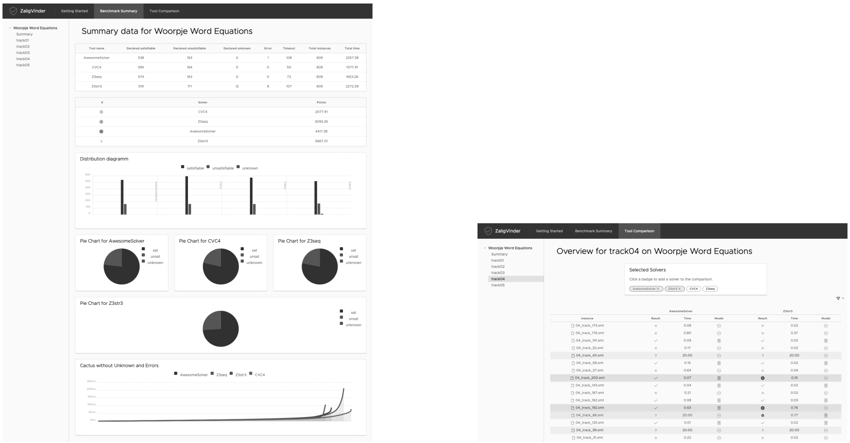
Endpoint	Additional get parameters	Functionalities
solvers	–	List of solver names
tracks	–	List of tracks including name, id, corresponding benchmark set, and all instances
tracks/ids	–	List of all track ids
tracks/groups	–	List of all benchmark set ids
tracks/info	–	List of all benchmark sets, their track ids and names
instances	–	List of all instances including their name, id, and expected result
instances/ids/track/<trackID>	–	List of all instances ids of track trackID
instances/ids/group/<benchmarkID>	–	List of all instances ids of benchmark group benchmarkID
instances/solvers/<instanceID>	solvers	Dictionary mapping the instanceID to a solver's result dictionary holding timeouted, result and time
instances/<instanceID>/model.smt	–	Shows the input formula of instance instanceID
results	–	List of all results including solver name, instanceid, and a Result dictionary holding timeouted, result and time
results/<trackID>	–	Restricts the results to track trackID
results/reference/<instanceID>	–	Dictionary holding expected result and satisfying solvers and unsatisfying solvers for instance instanceID
results/<solverName>/<instanceID>/output	–	Shows solverNames output on instance instanceID
results/<solverName>/<instanceID>/model	–	Shows solverNames model on instance instanceID
summary/<solverName>	–	Show the accumulated results for solverName holding timeouted, satisfied, unsatisfiable, error, unknown, time, and total instances
summary/solverName/trackID	–	Show the accumulated results for solverName on track trackID holding the summary data
chart/cactus	format (e.g png), solver names	Shows a cactus plot in the given format (default: plain jSON) for solvers (default: all solvers)
chart/distribution/<benchmarkID>	–	Shows a distribution diagram for <BENCHMARKID>
chart/keywords	format (e.g png)	Shows a barchart with accumulated keywords in the given format (default: plain jSON)
chart/scattered	solvers, format (e.g png)	Prints a scattered plot of two solvers
ranks/trackID	–	List of solver name and Par2 score points for trackID

REST API

The API offers an easy way to processing the data generated by ZaligVinder externally. In Table 5.1 we review the basic functionalities of the API.

We offer endpoints to the most general information being present within the database such as solver information, track and instance details, as well as a refurbished look on the results of the considered run e.g. summary data of a run. Next to these features we offer an access to various chart representations – namley cactus, distribution and scattered plots – of the results. The user can either access the plain data or an image rendered by using `matplotlib`. An unrelated but interesting view in regards to string solving is the ability of accessing a chart visualising the most used string functions within a run by visiting `chart/keywords`.

As ZaligVinder was build to be highly extensible, the implemented set of API endpoints forms only a start without limits to enhancing it.



Summary data for Woorpje Word Equations

Tool name	Declared satisfiable	Declared unsatisfiable	Declared unknown	Error	Timeout	Total instances	Total time
AwesomeSolver	538	163	0	1	108	809	2257.38
CVC4	595	164	0	0	50	809	1077.91
Z3seq	574	163	0	0	72	809	1653.26
Z3str3	519	171	12	8	107	809	2272.39

#	Solver	Points
1	CVC4	2077.91
2	Z3seq	3093.26
●	AwesomeSolver	4417.38
4	Z3str3	5667.01

a. Benchmark summary view

Overview for track04 on Woorpje Word Equations

Selected Solvers

Click a badge to add a solver to the comparison.

	AwesomeSolver			Z3str3		
Instance	Result	Time	Model	Result	Time	Model
04_track_173.smt	×	0.08	⊖	×	0.02	⊖
04_track_176.smt	×	0.80	⊖	×	0.37	⊖
04_track_141.smt	✓	0.09	▤	✓	0.02	▤
04_track_52.smt	×	0.17	⊖	×	0.02	⊖
04_track_40.smt	?	20.00	⊖	?	20.00	⊖
04_track_58.smt	✓	0.15	▤	✓	0.02	▤
04_track_37.smt	×	0.64	⊖	×	0.04	⊖
04_track_200.smt	✓	0.07	▤	⊗	0.19	⊖
04_track_143.smt	✓	0.04	▤	✓	0.02	▤
04_track_187.smt	×	0.21	⊖	×	0.02	⊖
04_track_162.smt	✓	0.08	▤	✓	0.03	▤
04_track_192.smt	✓	0.63	▤	⊗	0.76	⊖
04_track_66.smt	?	20.00	⊖	✔	0.77	▤
04_track_125.smt	✓	0.01	▤	✓	0.02	▤
04_track_99.smt	?	20.00	⊖	?	20.00	⊖
04_track_31.smt	×	0.22	⊖	×	0.02	⊖

b. Instance comparison view

Figure 5.2. Web GUI

GUI

To ease the comparison of different solvers, we implemented a basic graphical user interface (GUI) using the data of the created SQLite database. The GUI offers two categories to review the results: a benchmark summary view and a tool comparison view. The benchmark summary view – depicted in Figure 5.2 a – shows a summary for all benchmark sets and tracks. The sub navigation on the right hand side allows choosing a specific track or the summary for a whole set of benchmarks. Each page offers an overview table of grouped instances. The table contains the tool name, how many instances are declared as satisfiable resp. unsatisfiable, unknown instances (the solver terminates without a result before getting killed by the timeout limit), soundness errors (error), timed out instances, the total count of instances within a track resp. set of benchmarks, and the overall solving time. The second table shows a ranking for each solver participating in a track resp. benchmark set. The grading is easily modifiable and currently following a modified par2 score which is calculated using the results r_s for a specific solver s as follows

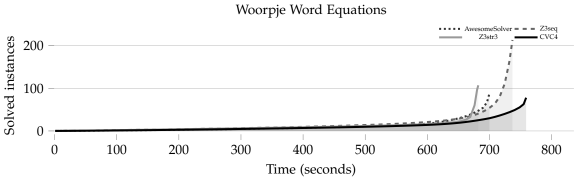
$$\text{par2}(s) = \sum_{i \in c} i_{\text{time}} + \sum_{i \in t} 2 \cdot \mathsf{TO} + \sum_{i \in e} 5 \cdot \mathsf{TO}$$

where $c \subseteq r_s$ is the amount of correctly solved instances, $t \subseteq r_s$ are instances declared as unknown or timeout and e being misclassified instances. TO is the predefined timeout.

The first diagram shows a distribution for each solver distinguishing between satisfiable, unsatisfiable and unsolved (timed out and unknown) instances. The next set of diagrams show the same distribution as before but as a pie diagram to ease identification in some cases. A cactus diagram follows. In these kind of plots all instances are sorted by their solving time and listed ascending as a point within a line diagram. The cactus plot lists all instances of a track resp. benchmark set excluding instances where a solver was not able to return an answer. It gives an intuition of how quick a solver comes up with the correct answers. By clicking on a label of the graph, the user is able to active resp. deactivate a specific solver.

The second view – the tool comparison view, depicted in Figure 5.2 b – offers the opportunity to compare different solvers per instances. This helps finding out what instances are causing unwanted behaviour for a particular solver. The navigation between different tracks and benchmark summaries is done by using the side navigation as explained previously. The top box allows as before to choose between the available solvers. Clicking a label activates or deactivates a solver. Solvers highlighted in green are part of the current comparison, whereas white labelled solvers are not.

The comparison table contains the instance name (corresponding to the input file name). A click reveals the definition of the input formula. The following columns are repeated for each solver being part of the comparison. We display the classification of a solver including a judgement whether a classification is correct by an icon. The ✔ indicates that a solver classified an instances as satisfiable whether ✖ indicates its unsatisfiability. An instance marked with **?** means that the corresponding solver was unable to identify an answer within the given timeout. Whenever an instance was correctly declared as satisfiable the model is available by clicking 🗎. An incorrect model is indicated by ⊞, whereas the absence is



	AwesomeSolver	Z3seq	Z3str3	CVC4
sat	538	574	519	595
unsat	163	163	171	164
unknown	0	0	12	0
timeout	108	72	107	50
soundness error	1	0	8	0
program crashes	0	0	0	0
Total correct	700	737	682	759
Time (s)	2257.38	1653.25	2272.39	1077.90
Time w/o timeouts (s)	97.38	213.25	132.39	77.90

Figure 5.3. Automatic table and cactus plot generation of ZaligVinder.

marked by ⊖.

Another feature is a highlighting of particular rows. Light blue rows mark instances where no solver being part of the comparison was able to find an answer. Light green rows mark instances where only on solver of the comparison was able to return an answer, whereas the witnessing solver is marked with a bold icon ✔, respectively ✖. A red highlighting indicates the presence of one solver which either misclassified an instance or produced a wrong model. The corresponding solver is again highlighted with a bold icon ✔, respectively ✖.

Commandline Tools

Another advantage of storing the data in an SQLite database is that it can be used for generating tables and plots for papers. As an example of this, Figure 5.3 was automatically generated by ZaligVinder. We provide a Python script – called `tablegen.py` – based on the package `npyscreen` allowing the generation of multiple visuals for external usage. After selecting a ZaligVinder database file and an output file name, the user selects benchmark groups and solvers being part of the external visuals. Currently we offer three selectable techniques:

1. summary tables of the results in LaTeX,

2. cactus plots of the results in LaTeX using TikZ, and

3. a summary page including cactus plots in AsciiDoctor [7] format.

Table 5.2. Summary of the collected benchmarks including the most used string operations

Benchmark Name	#Sets	#Instances	Most used string operations (Avarage occurence)				
			1st	2nd	3rd	4th	5th
Leetcode	43	2666	`len` (44.31)	`at` (25.3)	`indexof` (10.06)	`substr` (9.2)	`++` (2.93)
PyEx	57	25421	`len` (361.29)	`substr` (349.18)	`indexof` (334.44)	`contains` (34.17)	`++` (20.46)
AppScan	1	8	`to.re` (9.5)	`++` (1.75)	`suffixof` (1.12)	`in.re` (1.0)	`indexof` (1.0)
AutomatArk	2	19979	`to.re` (25.2)	`in.re` (2.25)			
BanditFuzz	1	357	`indexof` (3.31)	`len` (2.86)	`at` (2.36)	`replace` (2.21)	`substr` (2.1)
Cashew	1	394	`++` (13.59)	`to.re` (2.81)	`in.re` (2.3)	`len` (1.78)	
JOACO	1	94	`to.re` (26.26)	`++` (8.39)	`in.re` (0.81)	`len` (0.13)	`to.int` (0.05)
Kaluza	4	47284	`to.re` (33.53)	`++` (12.72)	`len` (6.94)	`in.re` (4.38)	
Kausler	1	120	`++` (197.88)	`substr` (10.27)	`len` (1.98)		
Trau Light	1	100	`++` (12.0)				
Norn	5	1027	`to.re` (14.88)	`in.re` (4.77)	`++` (2.77)	`len` (1.09)	
Pisa	1	12	`contains` (2.25)	`++` (1.08)	`substr` (0.83)	`replace` (0.83)	`len` (0.67)
Sloth	1	40	`in.re` (0.97)	`++` (0.93)	`to.re` (0.82)	`replace` (0.42)	`replaceall` (0.17)
Stranger	1	4	`to.re` (78.0)	`++` (13.0)	`in.re` (1.0)		
StringFuzz	17	1065	`++` (66.2)	`to.re` (41.39)	`at` (33.61)	`len` (32.05)	`in.re` (1.38)
StringFuzz Regex Generated	7	4170	`to.re` (160.99)	`in.re` (6.25)	`len` (0.37)		
StringFuzz Regex Transformed	2	10682	`to.re` (1.88)	`in.re` (1.41)	`len` (0.74)	`to.int` (0.2)	`++` (0.15)
Woorpje	5	809	`++` (43.04)	`len` (0.95)			
Z3str2	1	243	`++` (1.19)	`len` (0.6)	`to.re` (0.38)	`contains` (0.29)	`in.re` (0.25)
Total	152	114475	`len` (75.36)	`substr` (68.96)	`indexof` (66.94)	`to.re` (25.46)	`++` (11.19)

5.2 Benchmark Sets

We searched the literature to identify existing benchmarks, used by the different string solvers. These benchmarks have been incorporated into our tool ZaligVinder. We first overview their origin and briefly discuss which constraints are used within each of them and summarise the details in Table 5.2.

Leetcode Wei-Cheng Wu used the concolic execution engine Conpy [31] and PyExZ3 [13] to generate 43 different sets consisting of a total of 2666 instances. The basis are interview questions from the website `http://leetcode.com`. The length constraint heavy set consists of challenging 881 satisfiable and 1785 unsatisfiable instances forming a good regression test to make sure the interleaving theories of an SMT solver behave correctly.

PyEx Reynolds et al. [97] used the tool PyEx [13] - a symbolic executor for Python programs - to generate a set of 25,421 benchmarks. They used 19 target functions sampled from four popular Python packages to generate the resulting benchmark set.

AppScan Zheng et al. [123] generated a second set of benchmarks using the output of security warnings generated by IBM Security AppScan

Source Edition [64]. They ran the tool on popular websites to obtain traces of program statements which where translated into SMT formulae. The traces reflect potentially vulnerable information flows and therefore represent common real-world constraints. The set consists of 8 instances containing string functions and disequality constraints over strings.

BanditFuzz BanditFuzz [102] is a reinforcement learning driven fuzzing system for SMT solvers. The authors share a set of string benchmarks being generated by using their tool. The resulting 357 instances heavily containing the `indexof` predicate which makes them unique within our gathered instances.

Cashew Brennan et al. [28] used their tool called Cashew to normalise (in terms of their tool) 18,896 extracted benchmarks by Luu et al. [86] from the Kaluza benchmark set. By constructing this subset the authors aimed to eliminate the redundancy in the original set. The set varies between easy and difficult string constraints, with Boolean constraints, without using string operations.

JOACO In order to evaluate their tool JOACO, a tool to detect injection vulnerabilities, Thomé et al. [110] created a set of 94 instances based on 11 open-source Java Web applications and security benchmarks used in the literature. It displays a variety of instances containing string constraints, regular expressions and string operations.

Kaluza Saxena et al. [99] used their tool Kudzu, a symbolic execution framework for JavaScript, to generate more than 50,000 string solving problems. The instances were obtained by lowering JavaScript operations from real world AJAX web applications and are available on their website [100]. The instances are build around string constraints, membership in regular languages (given as regular expressions), and inequalities involving length constraints on string variables. While the size of the formulae varies per instance, the variety in the used string operations is rather small. The resulting set of benchmarks was translated by Liang et al. [81] into the SMT-Lib format. It uses string, boolean and linear constraints together with a small amount of string operations.

Kausler Kausler and Sherman [69] generated a set of benchmarks to evaluate string constraint solvers in terms of symbolic execution. The set was brought down to 120 instances by Thomé et al. [110]. It contains constraints from 8 Java programs via dynamic symbolic execution, aiming for real word application. The set mostly contains Boolean and string constraints without string operations.

Light TRAU Within this set of benchmarks generated by Abdulla et al. [2] each instance holds multiple easy, mostly unsatisfiable formulae consisting only of string constraints. The set aims for testing the ability of declaring inputs as unsatisfiable, which is in general harder than finding a solution.

NORN Abdulla et al. [4] share a set of 5 tracks consisting of queries generated during verification of string-processing programs [5]. Each formula is rather small compared to those in other sets of benchmarks, but makes heavy use of regular expressions containing Kleene stars. This makes it a challenging one for all solvers.

PISA Zheng et al. [123] generated a set of benchmarks using constraints from real-world Java sanitizer methods which where used to evaluate the PISA system [108]. It contains 12 complex instances including multiple different string operations like `indexOf`, `substring` as a result of the sanitiser structure.

SLOTH The string solver SLOTH [62] handles the so called straight line fragment of string constraints, that is essentially a formula which corresponds to a sequence of program assignments in SSA form including the assertion of regular constraints [84]. Their regression test suite contains 40 instances including the corresponding string operations. However, string solvers like CVC4 and Z3STR3 are not able to handle the present `str.replaceAll` function which is present in 7 instances and allows only using 33 out of 40 instance for a comparison between all solvers.

STRANGER Yu et al. [120] used this set of 4 real-world PHP web applications to evaluate their tool STRANGER - a tool detecting and sanitising vulnerabilities in PHP applications. Thomé et al. [110] manually translated these instances into the SMT-LIB format. The result was a set containing string operations, regular expression membership constraints, and string constraints.

StringFuzz Blotsky et al. [27] introduced a tool called StringFuzz to generate and transform SMT-LIB instances of string problems implemented in Python. The authors share a set of benchmarks generated using their tool, which aims to address typical industrial instances, potentially challenging for solvers. The set contains 17 tracks ranging from instances containing pure string constraints to hard to solve regular expression constraints. They aim for generating instances which follow structures hard to handle by some solvers (e.g. tree-like instances).

Z3str2 This set by the authors of Z3str2 [121] was initially generated as a regression test for their tool. Nowadays it seems to have a broader audience due to the evenly distributed occurrences of multiple string and regular expression predicates. The set consists of a single track holding 243 instances in total.

Within our work on string solvers, we extended this collection of benchmarks by the following sets.

Woorpje We created a set of benchmarks to test the abilities of our approaches. The set contains 5 tracks with instances containing mostly word equations, but also linear length constraints. Running this set on their competitors revealed its difficulty. The set is generated using several hard involved examples developed in the theoretical study of word equations. The first track (I) was produced by generating random strings, and replacing factors with variables at random, in a coherent fashion. This guarantees the existence of a solution. The generated word equations have at most 15 variables, 10 letters, and length 300. The second track (II) is based on the idea in Proposition 1 of [43], where the equation

$$\begin{aligned} & X_n a X_n b X_{n-1} b X_{n-2} \cdots b X_1 \\ = {} & a X_n X_{n-1} X_{n-1} b X_{n-2} X_{n-2} b \cdots b X_1 X_1 b a a \end{aligned}$$

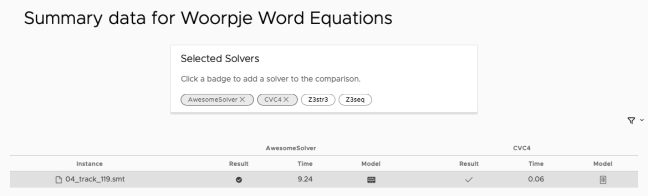
is shown to have a minimal solution of exponential length w.r.t. the length of the equation. The third track (III) is based on the second track, but each letter *b* is replaced by the left hand side or the right hand side of some randomly generated word equation (e.g., with the methods from track (I)). In the fourth track (IV) each benchmark consists of a system of 100 small random word equations with at most 6 letters, 10 variables and length 60. The hard aspect of this track is solving multiple equations at the same time. Within the fifth track (V) each benchmark enriches a system of 30 word equations by suitable linear constraints, as presented in this paper. This track is inspired by the Kaluza benchmark set in terms of having many small equations enriched by linear length constraints. All tracks, except track II which contains 9 instances, consist of 200 benchmarks.

AutomatArk In the need of regular expression heavy benchmarks we generated SMT-LIB benchmarks based on real-world regular expression queries collected by Loris D'Antoni. The set consists of two tracks – a simple and a hard track – having a total of 19979 instances. The simple track holds instances having a single regular expression membership constraint, and the hard track holds up to 5 membership constraints over a single variable per instance.

StringFuzz regex instances In 2020 we extended the StringFuzz suite by two regex heavy sets having a total of 15052 instances. The `StringFuzz-regex-generated` set contains randomly fuzzed instances only coping regular expressions and length constraints. The `StringFuzz-regex-transformed` set is the result of a transformation of instances supplied by Amazon Web Services related to security policies, being inspired by real-world input vulnerability violations.

5.3 Use cases

In this section we present two empirical studies on how to use ZaligVinder to debug a string solver. We highlight the abilities of the GUI to determine performance issues and soundness errors within the string solver of choice. Secondly, we demonstrate the extensibility of ZaligVin-



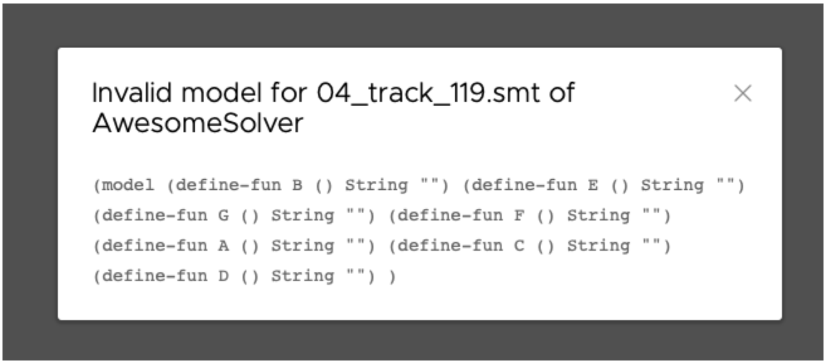
Summary data for Woorpje Word Equations

Selected Solvers

Click a badge to add a solver to the comparison.

	AwesomeSolver			CVC4		
Instance	Result	Time	Model	Result	Time	Model
04_track_119.smt	●	9.24	▤	✓	0.06	▤

Figure 5.4. Detailed analysis of ran instances. The line highlighted in grey indicates an error in a solver.



Invalid model for 04_track_119.smt of AwesomeSolver

```
(model (define-fun B () String "") (define-fun E () String "")
(define-fun G () String "") (define-fun F () String "")
(define-fun A () String "") (define-fun C () String "")
(define-fun D () String "") )
```

Figure 5.5. Generated models within ZaligVinders frontend

ders infrastructure and implement an analysis technique to discover more insights of our string solver and use this data for further external analysis.

5.3.1 Using the GUI

We are developers of the fictional solver AwesomeSolver and just finished implementing a new feature. To measure its performance we set up a run comparing against some of the state of the art string solvers, namely CVC4, Z3str3 and Z3seq, on the Woorpje benchmark set.
Following Section 5.1 allows us setting up a run easily since all competing solvers and the benchmark set are already preconfigured. Therefore, we create a new solver file following the template presented in Listing 5.3 and add all competing solvers as shown in line 8 to our new runner script.

Once the run finishes ZaligVinder automatically starts a web server showing the website being depicted in Figure 5.2. The presented diagrams indicate an insufficient performance of our string solver. Moreover, the summary table reveals an error within our solver.

We change our view to the *Tool Comparison* website and select appropriate filters as presented in Table 5.1.3. ZaligVinder presents the summary being depicted in Figure 5.4: our solver produces an invalid model on one instance. The produced model is revealed as soon as we click on the brick wall icon (Figure 5.5). Our solver tries substituting all variables by the empty word. Quickly comparing against the successfully validated model of CVC4 by clicking on the model icon highlights our wrongly presented solution further. The instance indicating the wrong model production is now used for debugging AwesomeSolver.

5.3.2 Extending the API for an individual post analysis

Adding a completely new individual analysis to ZaligVinder requires three modifications:

- ▻ setting up appropriate database queries,
- ▻ creating a controller for handling the results being returned by the database queries, and
- ▻ adding an endpoint to the webserver.

We are interested in getting all file names where a particular solver produced a wrong model. To add this view to our API, firstly we come up with the database query to our model. To achieve this we add the function implemented in Listing 5.5 to our SQLite interface located in `storage.sqlite`.

```
def getUnverifiedSATInstances(self,solver):
    query = '''SELECT Result.instanceid,
        ↪ TrackInstance.filepath FROM Result,
        ↪ TrackInstance WHERE Result.solver =
        ↪ ? AND Result.result = true AND
        ↪ Result.verified = false AND Result.
        ↪ instanceid = TrackInstance.id'''
    rows = self._db.executeRet (query,(solver,)
        ↪ )
    return {t[0] : {"filepath" : t[1]} for t in
        ↪  rows}
```

Listing 5.5. Adding a new database query to the model `storage.sqlite`

We simply ask for all instances where a solver returned `SAT` but the verification procedure was not able to validate the model. Secondly, we specify our internal controller logic, which is done by adding the function given in Listing 5.6 to the results controller located in `webserver.controllers` ↪ `.results`.

```
    def getUnverifiedSATInstancesForSolver (
        ↪ self,params):
        return webserver.views.jsonview.
            ↪ JSONView (self._results.
            ↪ getUnverifiedSATInstances (
            ↪ params["solver"]))
```

Listing 5.6. Adding a new logic implementation to the results controller `controller.results`

The controller again is a simple task, since all that is needed is forwarding the database result to the included JSON view. The view translates the passed Python 3 dictionary into an appropriate JSON format. The last step is adding an endpoint. We simply add the line given in Listing 5.7

```
app.addEndpoint (webserver.routing.RegexMatch("
    ↪ my_analysis/unverified/(?P<solver>[^/]+)
    ↪ "),self._rcontroller.
    ↪ getUnverifiedSATInstancesForSolver)
```

Listing 5.7. Adding a new endpoint to `startwebserver.py`

to the file `startwebserver.py`. It listens to the URL `http://localhost:8081/my_analysis/unverified/SOLVERNAME`, where `SOLVERNAME` is an arbitrary string. Whenever this endpoint is called the previously implemented function of our result controller is called and the solver name passed on. Guiding our browser to the URL `http://localhost:8081/my_analysis/unverified/AwesomeSolver` returns in the running example of this section the following string `{"558": {"filepath": "/home/mku/wordy/models/woorpje/track04/04_track_119`

`.smt"}}` – our previously described instance where AwesomeSolver returns a wrong model.

5.4 Analysis of the presented Framework and the Benchmark sets

Up to this point we discussed many details on how to apply our framework to string solvers, what a user's output looks like, and how the generated data can be used for a custom post analysis. We also introduced 19 benchmark sets and briefly introduced their origin, as well as the used operations. In this section we are looking at our framework and also the benchmarks from a different perspective and rate the quality and quantity of both.

5.4.1 Our Framework

We built the tool ZaligVinder to solve an issue when comparing our own implementations of string solving algorithms with respect to the performance of already published tools. While searching the literature we were not able to find a tool which is able to reliably compare string solvers with their special needs as for example validating models. To this extend we designed ZaligVinder tailored towards string solvers with the goal of not only comparing string solvers but also to ease the whole development procedure. As discussed in this chapter, our tool features many different techniques to identify performance issues and soundness errors in a user friendly way. We not only designed our framework to ease extensibility but also to easily perform a sufficient post analysis including several mechanism to export the data into a LaTeX article.

Given these facts we evaluate our framework based on the following research questions: 1. Are we able to reliability compare string solvers? 2. Are we able to identify bugs within string solvers? 3. Are we able to gather insights about the string solvers performance? 4. Are the post-processing mechanisms sufficient?

Table 5.3. Comparison of CVC4, Z3SEQ, and Z3STR3 on all benchmarks.

	CVC4	Z3SEQ	Z3STR3
sat	76466	76627	52464
unsat	30318	30070	24271
unknown	64	61	33506
timeout	7627	7717	4234
soundness error	0	7	9
program crashes	0	9	9
Total correct	106784	106690	76726
Time (s)	178364.27	215301.37	93145.69
Time w/o timeouts (s)	25824.18	60961.37	8465.69

RQ 1: Are we able to reliability compare string solvers?

With the goal of comparing string solvers in an easy way we simply follow the steps explained in Section 5.1.1. Once the run is finished we are able to review the results within the Web-GUI, simply generate LaTeX tables and plots or generate an AsciiDoctor website. Whatever data visualisation we choose, we are able to see which solver has the best performance, identify potential performance issues, or review insights on certain misclassified instances.

In Table 5.3 and Figure 5.6 we visualise a run of the major SMT-solver binaries CVC4 1.8, Z3SEQ 4.8.10, and Z3STR3 4.8.10 on all of our collected benchmarks. Clearly, we are able to observe all required details of the run.

Overall we think our goal of building a tool to compare string solvers was achieved in this sense.

RQ 2: Are we able to identify bugs within string solvers?

One of the key requirements when building a solver is its reliability. To this extend a solver is supposed to be sound and stable. Since we are facing an undecidable theory, non of the solvers has a warranty for a terminating algorithm, which we dealt with by enforcing hardware limits (e.g. a timeout). While developing our tool WOORPJE our techniques prove to be extremely valuable. As an example when we extended WOORPJE

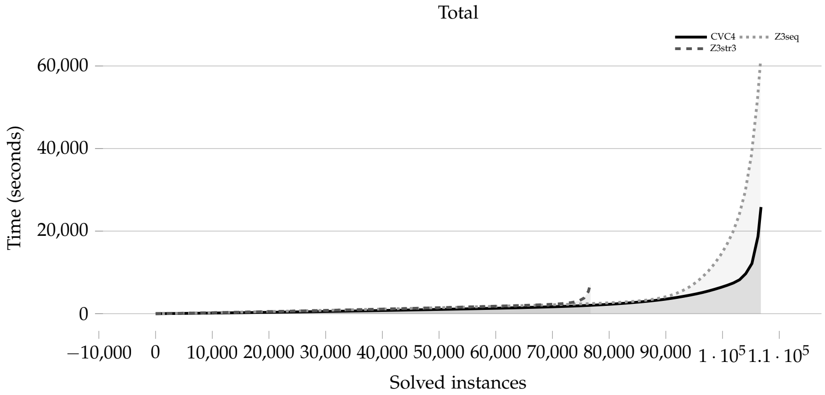


Figure 5.6. Cactus plot summarizing performance of CVC4, Z3seq, and Z3str3 on all benchmarks.

by a new simplification technique, ZaligVinder reported invalid models on a few amount at instances. Looking into the details presented within our Web-GUI revealed the instances on the actual error. Within this subtle error, variables where not fully substituted within the model. Having the abilities to review all affected instance made fixing this bug fairly easy.

Another interesting bug was revealed by looking at our competing solvers. Even though the solver developers agreed on the SMT-LIB 2.6 standard many issues especially related to Unicode support are not sorted out. As an example the solver CVC4 is using the escape sequence `\u{5c}` within the found model instead of using a backslash (\). The string theories within Z3 use the actual backslash symbol. Having said that, the solvers where not able to reason about the counterpart, and therefore reported an invalid model whenever the above situation occurs. To us, this is an unintended behaviour and needs to be resolved in future versions.

We not only discovered soundness issues but also many solver crashes during our tests. Looking again at Z3str3 within our tests we detected several segmentation faults on the considered benchmarks.

The run visualised in Table 5.3 and Figure 5.6 also reveals some solver errors. Z3seq classifies 7 instances of WoorpjeBench as unsatisfiable eventhough they are satisfiable. Z3str3 classifies 8 instances also on WoorpjeBench wrongly as unsatisfiable. Furthermore, it produces an invalid model on the Kausler benchmark suite.

RQ 3: Are we able to gather insights about the string solvers performance?

The design of the arm[2] selection method implemented within Z3str4 was heavily relying on empirical observations made by using ZaligVinder. As a short recap, Z3str4 probes which analyse syntactical structures of the input instances and upon that select a sequence of different solvers. To build the probes and sequences of solvers we used the insights gathered about the benchmarks in Section 5.2 and analysed the performance of each individual solver with respect to the higher order functions and relations, as well as occurrences of word equations, regular membership, and length constraints. Without having a framework like ZaligVinder this work would have been very tedious.

RQ 4: Are the post-processing mechanisms sufficient?

Our post-processing techniques introduced in Section 5.1.3 where developed incrementally while analysing string solvers. We started only having a rudimentary comparison view displaying the basic information of a run and extended it to visualising models, soundness issues, as well as the score of a solver according to the *par2* measurement. On of the latest addition is an extra table listing only the details of solver crashes and soundness issues to ease debugging specific cases. The output of results was also refined over time. Within a cactus plot the user is able to choose between multiple different display modes. The LaTeX summary tables themselves where extended to contain many required inputs to ease comparing string solvers also purely based on the article. In total, we are certain that the current setup features all required tools. Secondly, the incrementally process of achieving the perfect analysis shows the extensibility of our framework.

Looking at our framework from the aforementioned perspectives highlights the achievement of our requirements. Nevertheless, we believe that

[2] An arm is referred to a set of different string solvers being executed in a sequence.

there are still many features we can add to make the usability even better.

5.4.2 Benchmark Sets

The SMT-LIB standard in version 2.6 was released in early 2020 [111], and, next to introducing new high-level string operations such as `str.to_code` and `str.from_code` and renaming of operations, it introduces Unicode character support. However, many state of the art string solvers still only support the former version 2.5. Adding axioms for new functions or support for a different naming schema is not the biggest issue the solvers have to this end, but moving to Unicode support involves deep modifications of the SMT-solver core. Making steps into this direction requires manpower which, after optimising, debugging and analysing many different solvers, does not seem to be of highest priority: nearly all state-of-the-art string solver contain a multitude of soundness, parser and other sorts of errors. Analysing the benchmark sets introduced in Section 5.2 immediately let to the following research questions: 1. Are the gathered instances usable for all solvers supporting SMT-LIB? 2. What can we say about the general quality of the benchmark sets with respect to uniqueness, covered cases, and real life applications? 3. Can we reliably use these instances for testing string solvers?

Within this section we will describe the analysis of our benchmarks and furthermore perform steps to targeting raised issues.

RQ1: Are the gathered instances usable for all solvers supporting SMT-LIB? Even though, all collected instances are supposed to follow at least the unofficial SMT-LIB 2.0 standard, many instances are not parsable by all solvers, nor follow the conventions. Since within SMT-LIB we are facing the quantifier free fragment of string constraints, there should not be a single quantifier. The instances reveal a different view: some of them contain quantifiers. An exceptional example are the benchmarks of the Norn benchmark suite. Many of them quantify integer variables. Therefore, parsers as the one being implemented by the developers of CVC4 simple reject such an instance. Another example of insufficient designed instances are the Z3str2 regression tests. Within these instances we observe several occurrences of wrongly parametrised `indexof` functions

Table 5.4. Hash duplicates within the collected benchmarks

Benchmark Name	#Instances	#Duplicates
Leetcode	2666	14 (0.53%)
PyEx	25421	2864 (11.27%)
AppScan	8	0 (0.0%)
AutomatArk	19979	6626 (33.16%)
BanditFuzz	357	0 (0.0%)
Cashew	394	2 (0.51%)
JOACO	94	26 (27.66%)
Kaluza	47284	32712 (69.18%)
Kausler	120	7 (5.83%)
Trau light	100	0 (0.0%)
Norn	1027	0 (0.0%)
Pisa	12	0 (0.0%)
Sloth	40	0 (0.0%)
Stranger	4	0 (0.0%)
StringFuzz	1065	220 (20.66%)
StringFuzz Regex Generated	4170	59 (1.41%)
StringFuzz Regex Transformed	10682	4379 (40.99%)
Woorpje	809	0 (0.0%)
Z3str2	243	2 (0.82%)
Total	114475	46911 (40.98%)

(e.g. `(str.indexof X "ab")`). As we discussed in Section 2.4 as well as in Section 3.3 the function `indexof` takes three parameters, two patterns over $\mathtt{Pat}_A$ and an integer. Within these instances the developers missed the integer parameter, such that most solvers cannot handle the query. There are other examples of not well defined higher level functions within the set of benchmarks such that a proper usage requires an identification of insufficient definitions.

RQ2: What can we say about the general quality of the benchmark sets with respect to uniqueness, covered cases, and real life applications? Most of the gathered sets stem from solver developers trying to showcase the performance of their solver in a particular setting. There are only a few exceptions as we also pointed out in Section 5.2. In Table 5.2 we review the most used string operations within these benchmarks. Notably, an operation directly enforcing the correct handling of word equations, namely SUBSTR, is not the most prominent operator in these string benchmarks. In general, if a solver developer considers all sets of benchmarks for testing the abilities of a string solver, we think that our set forms a sophisticated basis. All commonly used operators are present and the instances are diverse in their own rights, again, when considering all sets. Nevertheless, when analysing all sets with respect to redundancy we cannot count on these instances. We performed an obvious comparison over all sets by simply computing the MD5 hash value of an instance. These results are visualised in Table 5.4. This simple method shows that out of 19 sets 11 sets contain duplicates. Moreover, out of 114475 instance more than 40% are duplicates. Overall, this is a huge drawback on the gathered instances and also questions the empirical evaluations reported within the publication of a certain benchmark set.

RQ3: Can we reliably use these instances for testing string solvers? We analysed the set of gathered benchmarks with respect to the standardised notion `(set-info :status` x`)` where x is either `sat` or `unsat`. We discovered that 47,534 instances where annotated with a *possible* result. A deeper look revealed that 13,725 instances were, in fact, wrongly tagged – meaning again only 33,809 instances were correctly annotated. To identify a wrongly classified instance we used the three leading SMT-solvers, namely CVC4, Z3STR3, and Z3SEQ, and ran them on these instances. If one of the string solvers was able to produce a model, we validated it by using the other competitors. Afterwards, we compared the verified results with the ones being present within the instance. In total, this high count of misclassified instances made the prior knowledge unusable, and proved the benchmarks as not being a reliable source for testing string solvers.

In the following we introduce a solution to the detected issues such that solver developers can safely use our modified set of the present benchmarks within solvers supporting SMT-LIB 2.5.

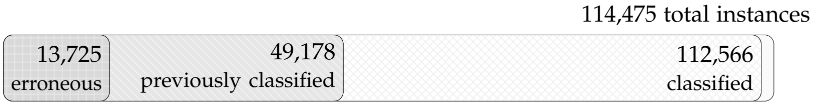


Figure 5.7. Visualisation of classified and cleaned-up instances

Solving some of the raised issues

To both ease the testing progress of string solvers as well as to make a step forward to implementing the new standard, we reconsidered the benchmark sets introduced in Section 5.2 and

A. structure the instances and remove or, respectively, replace non-standard operations to achieve parsability by the common state-of-the-art string solvers as well as readability by humans, and

B. annotate the benchmarks with reliable results.

C. remove identified duplicates.

We analysed the set consisting of 114,475 benchmarks and were able to annotate 112,556 instances by using the standardized notion `(set-info :status` x`)` where x is either `sat` or `unsat`. Overall we extracted 82,271 satisfiable and 30,285 unsatisfiable instances. During the analysis we discovered that 65,297 instances had no known result – meaning that 49,178 instances where annotated with a *possible* result. As pointed out earlier 13,725 instances were wrongly tagged. The high count of misclassified instances made this knowledge unusable. We visualise this in Figure 5.7.

To clean up the instances we wrote an SMT-LIB parser based on the ANTLR grammar of Thomé [109] in Python. While parsing the input we removed partially unsupported commands. Consequently, we also replace all SMT-LIB 2.6 operations by the widely supported SMT-LIB 2.5 operations. This is possible, since non of the analysed sets contains any of the newly introduced high-level string operators. To increase human readability we added handy line breaks and folded declaration and assertion part of the instances.

To validate a result in a satisfiable case we made sure at least one solver returns a valid model, which was verified by substituting the computed model into the original instance, and checking if we obtain a trivially satisfiable formula (for a valid model) or not (for an invalid model). Since the theory of strings is, in general, an undecidable problem, the verification of unsat-cases is not fully achievable. Thus, we added instances where no state-of-the-art solver produced a solution, and most solvers agreed on unsatisfiability. Moreover, we analysed the unsat-proofs produced by some of these solvers, and it is our intention to expand our approach to (partially) check and assert how reliable some of these proofs are.

The hash equivalent duplicates we identified in the previous section where simply removed from the respected sets. Therefore, the benchmarks are at least free of obvious redundant cases. In the future we plan to deepen this analysis by transforming each instance in some sort of a normal form. As a start we plan to rename all variables and sort each sub-formula by a predefined measure.

Overall the structured set obtained in this way eases the testing process for string solvers and offers new challenges to look at. On the practical side, these instances can – now having correctly annotated results – be used for a learning based approach to solving string constraints. As the sat-cases are correctly labelled, and the unsat-cases are obtained via the agreement of the existing solvers, we could hope to obtain, based on this training data, a solver which is at least as reliable as the existing ones. On the theoretical side, cleaned and human readable structured input offers a chance of identifying new sub-theories of string constraints which might not only form a decidable fragment, but also lead to more efficient algorithms to use for real-world problems, similar to the analysis seen in Section 4.3.

5.5 Related Work

Due to the amount of different string solvers it is surprising that we have not found a single tool incorporating the needs of comparing, debugging and analysing string solvers. The literature around string solvers mostly reports empirical data based on custom scripts which are not publicly available. A single tool which is also used for evaluating solvers at SMT-COMP is BenchExec [118]. A drawback of this tool is that it only features head-to-head comparisons of different solvers without being tailored towards string solving. As for SMT-COMP there is for example no validation of models. Secondly, since it is a pure benchmarking framework, it does not support the post-processing abilities our tool has to offer. Another closely related tool is called BenchKit [73] which is used for the Model Checking Contest [112]. The main drawback of this tool is again that it is primarily build to compare running times of solvers without allowing any further analysis. Secondly, it does not seem to be made to cope with SMT solvers.

With respect to the benchmark collection the SMT-LIB Initiative shares some sets of string solving benchmarks featured of the logics `QF_S` and `QF_SLIA` within their GIT repositories [105]. Meanwhile some of the explained benchmarks in this paper made it to their repository but not all of them. We plan to submit all collected sets after getting the permission of the respected authors to their repositories, too.

5.6 Conclusion

The set of string benchmarks now – for the first time – being available at a single place has been incorporated into our own, novel, benchmarking framework ZaligVinder. Next to this we made this set more reliable by annotating each instance with a result and improved human-readability of each instance. Having this huge database of cleaned up, annotated, and improved human-readable benchmarks is fundamental to the development and testing of novel and reliable string solving algorithms. From a theoretical point of view, this also offers a way to identifying new classes of string constraints, defined by, e.g., structural restrictions determined by the benchmarks, which could lead to algorithms to solve string constraints more efficiently.

Since 2020 when the first version of ZaligVinder was published in [76] our tool has proven to be reliable and easily extensible – not only due to being completely build in Python but also the easy ways to adapt the code structure we had chosen. Producing the output data of a virtual best solver, extracting all file paths of timed out instances, or printing the error messages a solver produced within a run had been some of the third-party extensions to ZaligVinder. One of the the most recent extensions is a swappable preprocessing mechanism. It permits the analysis and testing of the core of different string solving algorithms. Also the analysis of formulae, which were already simplified by different techniques, creates room for novel approaches within this field. Overall we still believe that ZaligVinder provides the basis for the development of an extended platform, to be used in comparing string solvers, in a more objective and complete manner.

In the future we plan to change the GUI infrastructure to provide a publicly available service. This gives the abilities to not only share the produced AsciiDoctor pages but also the complete visualised data with others. Secondly, an addition to perform a distributed run with ZaligVinder is a valuable extension to its core. This allows using not just one server but many to gather insights quicker than ever before.

Chapter 6

Analysing the Techniques

"Your face is drawn and ready for the next attack."

THE CURE

In Chapter 4 we developed three different techniques to solving string constraints. Each of these mechanisms approaches string constraints from a different perspective. Starting with a purely SAT based approach in Section 4.1, we went over to a rule-based strategy in Section 4.2. Both techniques centre word equations and allow additional linear constraints over the length of string variables. In the last section of Chapter 4 we used the ideas of a PSPACE-completeness proof to develop an algorithm for solving regular expression membership constraints, a sub-theory of string constraints. While each algorithm is interesting in its own rights, a technique has a higher impact if it is not only elegant but also applicable in practice, meaning an implementation which performs well on particular problems with respect to its competitors.

We devote this section to evaluate the performance and the practical impact of our techniques using the framework presented in Chapter 5. Note, that we do not replicate the data reported in the aforementioned publications but evaluate each technique within a homogeneous environment, against the most recent versions of string solvers. Since not only our results have been published a while ago but also developers of other solvers made improvements to their tools, there might be a variance within the results reported in this chapter.

6.1 Empirical Setup

In this section we briefly introduce the environment of our empirical evaluation. We start by introducing competing solvers and explain our choice. We move on to introducing the setup of the benchmarking framework and close this section by introducing the interpretation of our evaluation data.

6.1.1 Competing Solvers

We compare our techniques against six other leading string solvers available today. Our choice of competitors was influenced in multiple ways:

1. support for the theories of our approaches,
2. still maintained and available binaries, respectively compiling code,
3. support for SMT-LIB files.

Since we want to use the competing solvers throughout this section, our competitors have to support word equations, linear length constraints over string variables, and regular expression membership constraints combined with linear length constraints, as well as the related higher level functions and relations introduced in Section 2.4. We use CVC4 [80], a general-purpose SMT solver which reasons about strings and regular expressions algebraically. Secondly, we use Z3str3 [23], the latest mainline solver in the Z3-str saga. Thirdly, we compare against the second solver of the Z3 portfolio, namely Z3seq [106] or Z3 sequence solver. It was implemented by Nikolaj Bjørner and others at Microsoft Research, as part of the Z3 theorem prover. We also compare against the arm selection approach of Z3str4 which incorporates Z3str3 and Z3seq. We use the commit `b4e6d99` of the GIT repository `https://github.com/mtrberzi/z3.git`. The fifth solver, Z3-Trau [1] is also based on Z3 and uses an automata-based approach known as "flat automata" with both under- and over-approximations. Finally, we compare against OSTRICH [30] which uses a reduction from string functions (including word equations) to a model-checking problem that is solved using the SLOTH tool and an implementation of IC3.

We compare against CVC4's version 1.8 compiled from their official repositories, commit `59e9c87` of Z3str3, and the sequence solver, Z3-Trau, commit `1628747`, and OSTRICH version 1.0.1.

All of these tools support the full SMT-LIB standard for strings. We did not compare against the Z3str2 [122] or Norn [4] solvers as neither tool supports the `str.to_int` or `str.from_int` terms which represent string-number conversion, which are used in some sanitiser benchmarks, and are needed for the comparision within of our regular expression algorithm. Secondly, both tools are not maintained any more. Additionally, Norn does not support many of the other high-level string terms such as `indexof` or `substr` which are used in the benchmarks. The ABC [11] solver handles string and length constraints by conversion to automata. However, their method over-approximates the solution set of the input formula which may be unsound. Thus, we excluded ABC from our evaluation. We also were unable to evaluate against Trau [3] as the provided source code did not compile. We mention S3, Stranger, Sloth, Slog, Pass, Geocode, Hampi, and Kaluza in Section 3.3. These solvers where either excluded due to maintenance or lacking support for SMT-LIB input files.

6.1.2 ZaligVinder Setup

To evaluate our techniques we use the benchmark framework introduced in Chapter 5. We use a server running Ubuntu 18.04.4 LTS with two AMD EPYC 7742 processors and 2TB of memory. To ensure the correctness of the results reported by each solver, we use the validation technique introduced in Section 5.1.1. Each competing solver will also be used as validator to maintain fairness. For each instance we use a timeout of 20 seconds, which is a commonly used timeout for string solvers reported in multiple other publications.

6.1.3 Evaluation Data

Within this chapter we accumulate the results of each solver according to a track respectively benchmark set. The data we present corresponds to a single run. For a random single instance, the sample variance in execution time for 100 runs was 0.001 (0.07% of average execution time). Therefore, this fact is negligible. To visualise the data we use tables and cactus plots. Both are standard tools to visualise the performance of string solvers (cf. [21, 23, 24]).

For the tables each column corresponds to a particular solver while the rows are dedicated to specific data. The basic setup contains 9 labels. 1. *sat* corresponds to the count of instances reported as satisfiable by a solver, 2. *unsat* corresponds to the count of instances reported as unsatisfiable by a solver, 3. *unknown* corresponds to the count of instances where the solver stopped the search before the external limit (e.g. timeout) was reached, 4. *timeout* corresponds to the count of instances where the solver was stopped by an external timeout, 5. *soundness errors* correspond to the count of instances where a solver either produced a wrong model (validated by the other solvers) or reported unsatisfiability while another solver produced a correct model, 6. *program crash* corresponds to the count a solver terminated unexpected, 7. *total correct* reports the overall correctly solved instances, 8. *time* reports the overall solving time in seconds, and 9. *time w/o timeouts* removes all timed out instances from the above total time.

We mentioned cactus plots in Chapter 5 since ZaligVinder allows exporting these plots. As short recap, a cactus plot is a diagram visualising a solvers performance with respect to overall solved instances and the time it took. The x-axis visualises the number of solved instances while the y-axis measures cumulative time in seconds. Each solver is represented by a single line which corresponds to the number of solved instances in increasing order with respect to the time it took to solve it, meaning each point within a line of a solver corresponds to a solved instance and the cumulative time up to that point. A cactus plot is especially helpful to ease comparison of the performance of multiple different solvers; solvers with lines end further to the right and closer to the bottom of the plot have better performance. Note, within these plots we only visualise instances, where a solver returns a correct result with respect to the validation described previously.

In some experiments we use the virtual best solver among a set of solvers. The concept of the virtual best solver compares each instance of a run and picks the solvers which was performing best. The virtual solver can be seen as a portfolio approach of running all solvers in parallel while returning the result of the solver who finishes first with a correct result. We use this concept to visualise the maximum performance one might achieve by using the considered solvers.

6.2 Evaluation of the SAT based approach

We introduced a technique to solving string constraints restricted to the theory of word equations and linear length constraints in Section 4.1. We implemented this technique using `C++` and used Glucose 3.0 as underlying SAT-solver. In the following we will refer to the solver which implements this algorithm as WOORPJESAT, instead of just calling it WOORPJE to not confuse with other algorithms implemented within WOORPJE. The evaluation was performed on a newly created set of benchmarks called WOORPJEBENCH (see Section 5.2) for details, and a subset of the previously published *Kaluza* benchmarks. Since Kaluza is from a symbolic execution engine, the benchmarks are not pure word equations with length constraints but also involving variables with Boolean sorts, assignment to Boolean variables and disjunctions over expressions originating from the branching in programs. In order to use this benchmark set we thus developed a very naive interpreter for SMT into WOORPJE: the interpreter simply enumerates all the possible systems of word equations that can result from the SMT-file. These are afterwards checked for satisfiability, and if satisfiable the resulting system of word equations is output in SMT-format. These final output system of word equations is what we refer to as our KALUZABENCH set. The extraction procedure was run on the small and satisfiable set of the full KALUZA benchmark set resulting in KALUZABENCH consisting of 10853 system of word equations. The larger models contain features unsupported by WOORPJE thus they were left out from the extraction. These features are regular expression membership constraints, Boolean constraints, and extended functions like `contains`, `replace`, `substr` to name a few. Note, the set of benchmarks we used in [42] originally contained 14793 instances. We analysed these queries further and removed 3940 instances solely containing simple inequalities over integers, since we are interested in evaluating the performance of our

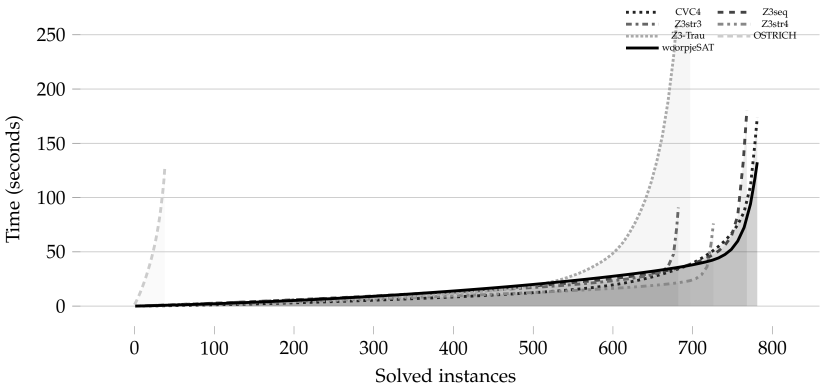


Figure 6.1. Cactus plot summarising performance on the complete WoorpjeBench set.

Table 6.1. Results for the complete WoorpjeBench set.

	CVC4	Z3seq	Z3str3	Z3str4	Z3-Trau	OSTRICH	woorpjeSAT
sat	617	604	519	562	560	34	632
unsat	164	164	171	164	188	5	149
unknown	0	0	12	0	0	768	3
timeout	28	41	107	83	61	2	25
soundness error	0	0	8	0	51	0	0
program crashes	0	0	0	0	0	0	0
Total correct	781	768	682	726	697	39	781
Time (s)	733.88	1000.59	2248.01	1736.09	1785.39	2402.1	638.66
Time w/o timeouts (s)	173.88	180.59	108.01	76.09	565.39	2362.1	138.66

algorithm to solving word equations.

6.2.1 Detailed results of the evaluation

In Table 6.1 and Figure 6.1 we visualise the cumulative results on WoorpjeBench set. Even though our technique was published in 2019 it is still heavily competitive. Together with CVC4 it solves the most cases (781 instances), but 0.15x faster overall and 0.25x faster removing timed out instances. Thereby, CVC4 solves more unsatisfiable instances and woorpjeSAT more satisfiable cases (15 instances each). Their closest follower is Z3seq solving 768 cases correctly. Overall woorpjeSAT solves 13 instances more and is 0.57x faster. The gap to the third successor Z3str4 is even

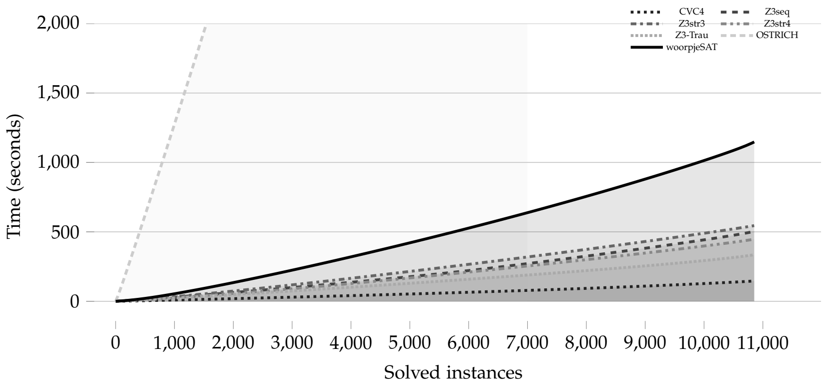


Figure 6.2. Cactus plot summarising performance on KaluzaBench.

Table 6.2. Results for KaluzaBench.

	CVC4	Z3seq	Z3str3	Z3str4	Z3-Trau	OSTRICH	woorpjeSAT
sat	10853	10853	10852	10853	10853	6998	10853
unsat	0	0	0	0	0	0	0
unknown	0	0	0	0	0	3855	0
timeout	0	0	1	0	0	0	0
soundness error	0	0	0	0	0	0	0
program crashes	0	0	0	0	0	0	0
Total correct	10853	10853	10852	10853	10853	6998	10853
Time (s)	147.25	503.59	564.85	447.04	335.29	14224.0	1147.57
Time w/o timeouts (s)	147.25	503.59	544.85	447.04	335.29	14224.0	1147.57

bigger: woorpjeSAT solves 55 instances more and is 1.72x faster. On the bottom end we find OSTRICH only solving 39 instances on the woorpje benchmark set. Not considering timeouts, woorpjeSAT is 16.04x times faster than OSTRICH.

The benchmark set also reveals 8, respectively 51, soundness errors for Z3str3, respectively Z3-Trau, while woorpjeSAT has none.

As explained in the corresponding Section 4.1, the woorpjeSAT mechanism to solving linear constraints is straight forward. If we only consider Track 1 to 4, woorpjeSAT solves 10 more instance 0.7x faster than CVC4, which again demonstrates the effectiveness of the approach to solving word equations. Detailed data on all tracks can be found in Appendix B.1.

The KaluzaBench is visualised in Table 6.2 and Figure 6.2. On this

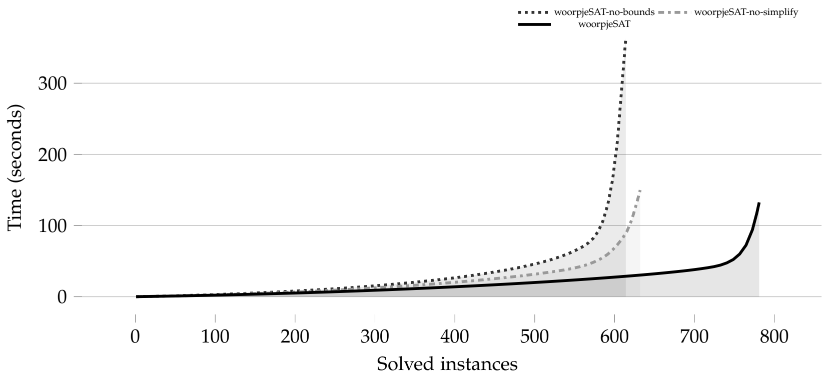


Figure 6.3. Comapring disabled sharpen the bounds and simplifier against all techniques on WoorpjeBench.

Table 6.3. Comapring disabled sharpen the bounds and simplifier against all techniques on WoorpjeBench.

	woorpjeSAT-no-bounds	woorpjeSAT	woorpjeSAT-no-simplify
sat	614	632	632
unsat	0	149	0
unknown	8	3	54
timeout	187	25	123
soundness error	0	0	0
program crashes	0	0	0
Total correct	614	781	632
Time (s)	4120.84	638.66	2888.51
Time w/o timeouts (s)	380.84	138.66	428.51

set all competing solvers, but OSTRICH and Z3str3 solve all instances. Unfortunately, woorpjeSAT is 6.79x slower than the leader of the set CVC4. The variance in running times compared to WoorpjeBench has two potential reasons: 1. the competitors benefit from their strong arithmetic components (most instances contain linear constraints), 2. specialised rewrite rules are used to simplify the instances of this previously known set of benchmarks (as e.g. seen in Section 5.1.2). As stated previously, in order to catch up, we have to improve our technique to solve linear length constraints over string variables.

To close this section we briefly compare the efficiency of the simplification rules on WoorpjeBench introduced in Section 4.1.5 and the bound refinement within our algorithm introduced in Section 4.1.3. We compare woorpjeSAT which uses these techniques against an implementation where we disable these mechanisms. By woorpjeSAT-no-simplify we refer to the solver not using any simplification steps, where woorpjeSAT-no-bounds names the version not using the search guidance technique. The results are given in Table 6.3 and Figure 6.3.

Overall woorpjeSAT is 3.52x faster than woorpjeSAT-no-simplify solving 149 more instances. The implemented search guidance pays off even more woorpjeSAT is 5.45x faster than woorpjeSAT-no-bounds solving 167 more instances. Next to the fact, that our purely SAT based approach is not able to determine the unsatisfiablity of an instance, this newly added techniques not only give an option to cope with this problem, but also speed up the satisfiable cases.

6.2.2 Conclusion

These refreshed experiments show even after two years that this purely SAT-based approach is still viable and competitive with state-of-the-art string solvers. The instances our solver solves, while the others do not, show that our encoding could be a beneficial extension in a portfolio approach to solving word equations. As such, it seems to us that our approach enriches in a non-trivial way the current landscape of string solving. A potential candidate could be the solver Z3str4 where woorpjeSAT could be used as one of the assisting solvers.

6.3 Evaluation of the Rule-based approach

The second algorithm was introduced in Section 4.2. The approach is based on Levi's lemma (see Section 3.5.2) and exhaustively applies the aforementioned rules to a system of word equations possibly enriched with linear constraints. The algorithm was implemented in C++ and potentially allows using CVC4, Z3str3, and Z3seq as assisting SMT-solvers. In the following we will refer to the implementation of this algorithm by woorpjeLevi instead of just calling it `woorpje`. For the evaluation we again use the benchmark set WoorpjeBench and the previously introduced subset of Kaluza called KaluzaBench. We used an additional set called StringFuzzBench which is shared by Blotsky et al. [27], who used their tool StringFuzz to generate instances. The previously presented set (see Section 5.2) contains higher-level operations which are not supported by our approach. We use their instances, restricted to the feature set woorpjeLevi is able to handle. They consist of instances named *concats*. The structure of the system of word equations is either based on right-heavy, deep tree concatenations of variables or balanced deep trees. Secondly they generate instances, which consider large length constraints and a mixture between length constraints and the previously mentioned concats. The third style of benchmarks holds instances which contain overlapping solutions within the substitutions of the variables. They share sets of satisfiable and unsatisfiable instances.

6.3.1 Detailed results of the evaluation

For the evaluation we wanted partly to compare each of the transformation system with assisting SMT-Solver with different heuristics and three different settings: the transformation system alone (without external SMT-solvers), against the 3 SMT-solvers (CVC4, Z3str3, Z3seq) alone, and the SMT-Solvers without guidance of the technique described in this paper. We ran these tools on each benchmark set with an overall timeout setting of 20 seconds for each system of word equations. The timeout for woorpjeLevi's calls to externals SMT-solver was set to 15 milliseconds. This limit seems tight. Since our goal is to minimize the input system of word equations to a form which is easy to handle by an SMT-Solver, the limit is still appropriate. We measured the time used by each solver[1] to reach a verdict, the verdict and the number of SMT-solver calls. If a solver times out we treat it as an "Unknown" verdict - although we will write it in the result tables as a timeout. Afterwards, we gathered all of the results and classified each instance as either satisfiable or unsatisfiable.

The individual results for WoorpjeBench, KaluzaBench, and StringFuzzBench are in Table 6.4, Table 6.5, and Table 6.6, respectively. By a superficial inspection of the listed results it is hard to get a definitive

[1] Here we refer to each instantiation woorpjeLevi as a different solver

Table 6.4. Summary benchmark data for WoorpjeBench (809 Benchmarks): (✔: correctly classified, ⌛: timeouted instances, 💣: incorrectly classified ⑂: calls to the external solver, 🕘: total time in seconds)

Heuristic Number		2			4			3			1			5	–
Parameters		314	1123	15	2	400	20	1.075	2	0.15	3.14	0.15	1.12358	–	–
	✔	748 (92.46%)	749 (92.58%)	757 (93.57%)	764 (94.44%)	707 (87.39%)	747 (92.34%)	746 (92.21%)	709 (87.64%)	760 (93.94%)	764 (94.44%)	762 (94.19%)	762 (94.19%)	707 (87.39%)	781 (96.54%)
	⌛	44 (5.44%)	39 (4.82%)	46 (5.69%)	44 (5.44%)	78 (9.64%)	45 (5.56%)	44 (5.44%)	77 (9.52%)	46 (5.69%)	44 (5.44%)	46 (5.69%)	46 (5.69%)	78 (9.64%)	28 (3.46%)
CVC4	💣	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
	⑂	900	552	1279	1957	79	1808	1451	77	1987	1886	2001	1962	78	–
	🕘	1216.4	1172.96	1111.77	1144.26	1908.76	1258.08	1273.25	1907.83	1221.66	1134.98	1156.13	1164.62	1911.38	732.98
	✔	757 (93.57%)	777 (96.04%)	749 (92.58%)	783 (96.79%)	769 (95.06%)	704 (87.02%)	782 (96.66%)	749 (92.58%)	710 (87.76%)	782 (96.66%)	783 (96.79%)	783 (96.79%)	708 (87.52%)	769 (95.06%)
	⌛	36 (4.45%)	26 (3.21%)	39 (4.82%)	25 (3.09%)	26 (3.21%)	81 (10.01%)	25 (3.09%)	46 (5.69%)	76 (9.39%)	26 (3.21%)	25 (3.09%)	25 (3.09%)	77 (9.52%)	40 (4.94%)
Z3seq	💣	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
	⑂	827	2044	328	2411	2196	81	2511	1057	76	2329	2344	2397	77	–
	🕘	1101.25	859.15	1251.33	962.1	948.7	1981.7	982.49	1266.6	1930.41	945.12	960.46	971.26	1920.54	1005.96
	✔	730 (90.23%)	742 (91.72%)	719 (88.88%)	751 (92.83%)	723 (89.37%)	703 (86.9%)	747 (92.34%)	708 (87.52%)	717 (88.63%)	748 (92.46%)	751 (92.83%)	751 (92.83%)	708 (87.52%)	684 (84.55%)
	⌛	62 (7.66%)	61 (7.54%)	67 (8.28%)	57 (7.05%)	69 (8.53%)	81 (10.01%)	60 (7.42%)	76 (9.39%)	75 (9.27%)	59 (7.29%)	57 (7.05%)	56 (6.92%)	75 (9.27%)	103 (12.73%)
Z3str3	💣	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	8 (0.99%)
	⑂	666	1199	312	1626	1884	81	1671	76	943	1661	1611	1608	75	–
	🕘	1688.45	1577.13	1818.98	1556.51	1876.46	2003.17	1620.49	1925.3	1895.0	1556.28	1563.71	1561.59	1907.98	2220.13

Table 6.5. Summary benchmark data for KaluzaBench (10853 Benchmarks): (✔: correctly classified, ⌛: timeouted instances, 💣: incorrectly classified ⑂: calls to the external solver, 🕘: total time in seconds)

Heuristic Number		2			4			3			1			5	–
Parameters		314	1123	15	2	400	20	1.075	2	0.15	3.14	0.15	1.12358	–	–
	✔	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)
	⌛	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
CVC4	💣	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
	⑂	26	3	2420	8649	0	73	1861	0	8685	8673	8682	8681	0	–
	🕘	421.4	454.63	401.32	454.37	466.78	465.83	449.46	473.95	460.32	449.45	449.9	450.45	459.29	316.09
	✔	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)
	⌛	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
Z3seq	💣	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
	⑂	10	350	1	8951	73	0	8991	1861	0	8916	8926	8929	0	–
	🕘	591.82	588.47	598.46	837.05	613.4	612.82	857.6	667.04	623.8	863.76	863.16	865.94	606.02	599.93
	✔	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10853 (100.0%)	10852 (99.99%)
	⌛	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	1 (0.01%)
Z3str3	💣	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
	⑂	15	2305	0	8458	73	0	8480	0	1861	8480	8481	8481	0	–
	🕘	611.23	652.45	628.58	823.76	616.56	611.91	836.79	625.94	656.05	852.46	853.24	852.13	613.41	605.39

picture of which heuristic strategy is better, and also in regards to which assisting SMT-solver is the best choice.

On WoorpjeBench heuristics 2, that is the invoking of an SMT-Solver whenever a system of word equations reaches a certain bound, is most helpful for Z3str3 with a bound of 2. Using this setup Z3str3 solves 67 more instances and is 0.43x times faster with an amount of 1626 external solver calls. For Z3seq heuristics 4, that is the ratio of newly introduced terminals by performing a transformation step, with a ratio of 0.15 is most helpful. Z3seq gains a speed-up of 5% and solves 14 more instances. Applying our technique to CVC4 is not beneficial. The best setup using our technique, is applying heuristics 4 with a ratio of 3.14. CVC4 is 0.55x faster without our technique and solves 17 more instances. It does however appear that the errors we observed earlier within Z3str3 on WoorpjeBench seem to be compromised by using our technique. This raises hope that the errors within Z3str3 might be easy to fix, since our rewrite rules avoid running into this trap. Recap, the given total times while using heuristics 5 vary between different solvers, because we are using the corresponding integer solver to verify linear length constraints.

Table 6.6. Summary benchmark data for StringFuzzBench (600 Benchmarks): (✔: correctly classified, ⧗: timeouted instances, 💣: incorrectly classified ⑂: calls to the external solver, ◷: total time in seconds)

Heuristic Number		2			4			3			1			5	–
Parameters		314	1123	15	2	400	20	1.075	2	0.15	3.14	0.15	1.12358	–	–
	✔	292 (48.67%)	293 (48.83%)	346 (57.67%)	358 (59.67%)	293 (48.83%)	302 (50.33%)	335 (55.83%)	292 (48.67%)	357 (59.5%)	357 (59.5%)	358 (59.67%)	359 (59.83%)	292 (48.67%)	539 (89.83%)
	⧗	84 (14.0%)	84 (14.0%)	94 (15.67%)	96 (16.0%)	83 (13.83%)	93 (15.5%)	84 (14.0%)	84 (14.0%)	94 (15.67%)	94 (15.67%)	94 (15.67%)	94 (15.67%)	80 (13.33%)	61 (10.17%)
CVC4	💣	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
	⑂	84	84	298	301	83	203	130	84	380	377	432	375	80	–
	◷	1943.28	1943.65	2180.83	2277.3	1998.64	2279.67	1943.89	1946.71	2234.09	2234.54	2235.77	2234.89	2013.37	2544.16
	✔	325 (54.17%)	407 (67.83%)	330 (55.0%)	403 (67.17%)	357 (59.5%)	328 (54.67%)	405 (67.5%)	366 (61.0%)	325 (54.17%)	409 (68.17%)	408 (68.0%)	403 (67.17%)	326 (54.33%)	231 (38.5%)
	⧗	84 (14.0%)	84 (14.0%)	84 (14.0%)	87 (14.5%)	86 (14.33%)	85 (14.17%)	88 (14.67%)	84 (14.0%)	84 (14.0%)	86 (14.33%)	88 (14.67%)	90 (15.0%)	79 (13.17%)	321 (53.5%)
Z3seq	💣	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
	⑂	84	680	84	389	86	85	390	128	84	399	384	401	79	–
	◷	2037.15	2167.8	2038.9	2220.59	2214.38	2059.97	2229.32	2021.04	2031.76	2221.11	2225.22	2247.99	2063.55	6709.65
	✔	326 (54.33%)	359 (59.83%)	325 (54.17%)	364 (60.67%)	320 (53.33%)	321 (53.5%)	367 (61.17%)	325 (54.17%)	356 (59.33%)	363 (60.5%)	362 (60.33%)	359 (59.83%)	320 (53.33%)	375 (62.5%)
	⧗	86 (14.33%)	105 (17.5%)	85 (14.17%)	103 (17.17%)	103 (17.17%)	84 (14.0%)	102 (17.0%)	84 (14.0%)	86 (14.33%)	105 (17.5%)	106 (17.67%)	107 (17.83%)	82 (13.67%)	222 (37.0%)
Z3str3	💣	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
	⑂	86	235	85	403	103	84	384	86	124	306	300	289	82	–
	◷	2034.28	2675.65	2035.83	2654.74	2568.03	2100.72	2667.25	2030.64	2044.83	2699.82	2719.21	2725.01	2094.29	4545.43

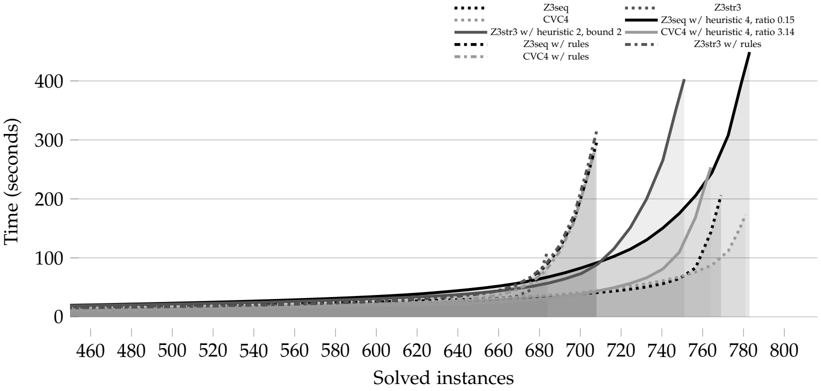


Figure 6.4. Cactus plot summarising performance on WoorpjeBench.

It should be noted that the table sometimes indicates an increase in solving time for an instance, whenever the count of SMT-solver calls goes up. In this cases the SMT-solver was usually not able to find a solution within the given timeout limits or simply was not able to draw any conclusion at all.

On KaluzaBench our technique does not seem to be helpful. All solvers except for Z3str3, which fails on one instance, solve all instances without our technique. Using heuristics 1, that is reaching a predefined depth within the graph induced by applying our rules, with a bound of 314, Z3str3 solves the remaining instance but is 1% (roughly 5 seconds) slower by applying our technique. A remarkable result is that the transformation without any external string solver is capable of classifying all of the KaluzaBench instances, and does so in competitive time to the fastest

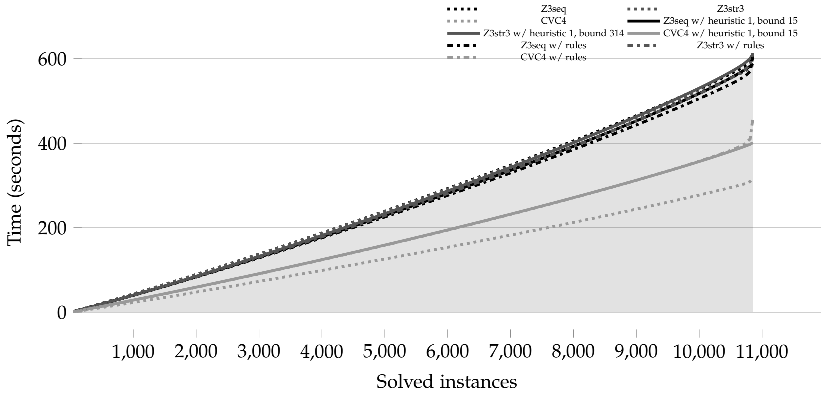


Figure 6.5. Cactus plot summarising performance on KaluzaBench.

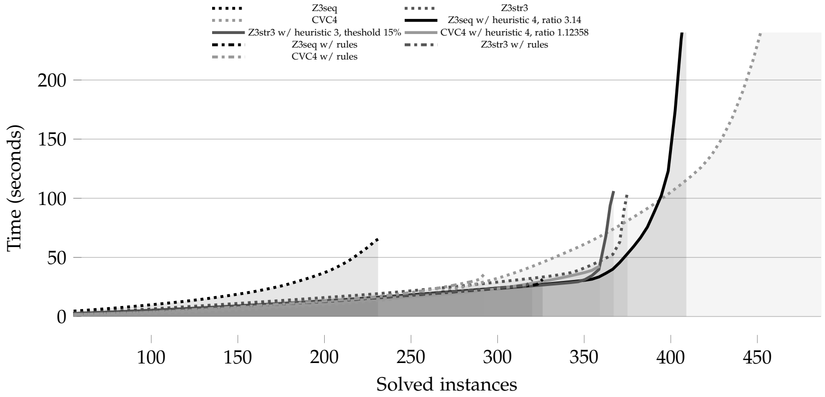


Figure 6.6. Cactus plot summarising performance on StringFuzzBench.

SMT-solver, whose time is given in the last column marked with heuristic – the technique presented in this paper is not used in that case.

The results look slightly better on StringFuzzBench. Our technique is not helpful for CVC4 and Z3str3 but for Z3seq. Using again heuristics 4 with a ratio of 3.14, Z3seq is 2.02x faster and solves 178 more instances. Z3str3 is performing best by applying heuristics 3, that is by applying one of our transition rules the length of the equation system increases quicker than a threshold of 15%. It is 0.7x faster but fails on solving 8 cases instances compared to Z3str3 without applying our technique. CVC4 gains a minor speed-up by using heuristics 4 with a ratio of 1.12358. This setup is 14% faster overall than the SMT-Solver itself but fails on solving 180 instances. These results are not strictly encouraging for our technique, as they show our technique sometimes reduces the number of correctly solved instances compared to the amount solved by each SMT-solver individually. However, we analysed StringFuzzBench in depth and observed that lots of equations in this benchmark set have a particular structure namely

$$\mathsf{X}_0 \doteq \beta \quad \text{and} \quad \mathsf{X}_0 \doteq \mathsf{X}_1\mathsf{X}_2\ldots\mathsf{X}_n,$$

where β is a sequence of terminals. Applying our rules to these examples result in a huge search space - and no matter which path is taken in the search space, a solution will be reached. The StringFuzzBench set thus constitute a malicious example to our rewrite rules which might explain the slowdown for CVC4.

We plot the results for WoorpjeBench, KaluzaBench, and StringFuzzBench showing each of the SMT-Solvers without our technique, the best performing technique, and purely using our rules where the SMT-Solver plays an assisting role in Figure 6.4, Figure 6.5 and Figure 6.6. Purely looking at these plots reveals the diversity of our benchmark selection. While in general CVC4 has the best performance in total, Z3seq is able to beat CVC4 on WoorpjeBench by using our technique. In general our technique seems to improve the performance of Z3seq on word equations such that integrating a rule-based solving strategy into its core is potentially useful. Another interesting observation is the performance of Z3str3 on StringFuzzBench. While having nearly no slowdown on the instances Z3str3 solves solely, it is able to beat Z3seq by using our technique. This observation might point to the structure of this benchmark set which seems to be harmful for the arrangement solving strategy Z3str3 uses. In Figure 6.7 we plot all solvers together with the virtual best solver over all heuristic runs using the corresponding solver as assisting SMT-solver. This plot clearly shows that our technique is a valuable addition to Z3seq and Z3str3. The related virtual best solver not only solves more instances but

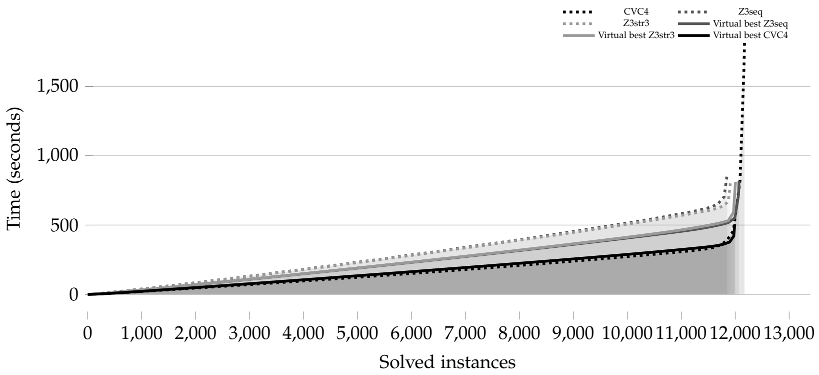


Figure 6.7. Cactus plot summarising performance on all benchmarks of the SMT-Solvers including the virtual best solver of all heuristic runs per solver.

also needs less time to do so. Note, in order to achieve the performance of the virtual best solver a natural next step is the detailed analysis of when a certain heuristic (with respect to the input query) performs best.

6.3.2 Conclusion

While the results we report in [42] emphasises our technique in a brighter light, the solver developers improved their solvers in such a way that it copes with similar techniques. Especially the developers of CVC4 extended their abilities from version 1.7 which we compared against, to the current version 1.8 which is used in this evaluation. Nevertheless, these observations suggest that our transformation techniques will be useful inside the string theory solver of existing SMT-solver - especially if it can be determined a priori what particular class of equations (from a structural point of view) can be handled by the SMT-solver most efficiently.

6.4 Evaluation of the procedure for solving regular membership constraints

The proof for the sub-theory of string constraints only containing regular membership predicates, concatenation, and length constraints in Section 4.3 directly lead to a decision procedure presented in Section 4.3.3. The resulting algorithm was implemented into Z3str3 by Berzish. Within the theory of strings we implemented an SMT-style sub-solver which now handles all formulae involving regular expression membership predicates. To not confuse, we refer to this implementation as Z3str3RE instead of calling it Z3str3. It is worth mentioning that parts of our sub-solver are already present in the mainline commit of Z3str3 we evaluate against in this section. To speed up our solvers performance we developed several heuristics to minimise the search space. In the following, heuristics which are implemented within mainline Z3str3 are marked separately.

The evaluation of Z3str3RE was performed on the benchmark set we analysed in Section 4.3.1. We restricted the set of benchmarks we gathered in Section 5.2 to 22425 instances containing at least one regular expression membership constraint. These are instances from AppScan, Cashew, Joaco, Norn, Stranger, Sloth, StringFuzz, Z3str2 regression, Kaluza. Additionally, we added instances from the BanditFuzz suite, which was obtained by private communication with the authors. Furthermore, we used the AutomataArk, StringFuzz-regex-generated, and StringFuzz-regex-transformed benchmarks, which are explained in detail in Section 5.2.

6.4.1 Additional Heuristics

As pointed out in Section 4.3.3 computing the intersection of our automata is extremely expensive. To this extend we introduce three heuristics which primarily try to completely avoid this step, or at least ease this process.

Lazy Construction of the Intersection Automata

To potentially avoid building the product automaton, we measure the costs of constructing an automaton based on a regular expression in advance.

Afterwards we lazily construct all automata based on this measurement and therefore, in case quickly determining the emptiness of an automata intersection (e.g. by building an automaton for a regular expression $\varnothing$), we might be able to completely avoid the expensive constructions. We assign the costs to a regular expression R using the following function $\mathsf{costs} : \mathtt{RegExC}_A \to \mathbb{N}$ inductively defined by

$$\mathsf{costs}(R) = \begin{cases} 0 & \text{if } R \in \{\,\varnothing, \varepsilon\,\}, \\ 1 & \text{if } R \in A, \\ \mathsf{costs}(R_1) + \mathsf{costs}(R_2) & \text{if } R = R_1 \diamond R_2 \text{ and } \diamond \in \{\,\cdot, \cup, \}, \\ 2 \cdot \mathsf{costs}(R_1) & \text{if } R = R_1^*, \\ (\mathsf{costs}(R_1))^2 & \text{if } R = \overline{R_1}. \end{cases}$$

Note, in case we are not able to determine an empty intersection of our automata, we construct the whole resulting intersection automaton to maintain soundness.

Prefix and Suffix Over-Approximation

Many of the regular expressions occurring in practice allow a simple inspection of their syntax in order to determine the emptiness of their intersection. From an implementation point of view it is hard to grasp the general structure of a regular expression but simple to analyse facts based on a fixed length. To this extend we determine the first letter of all regular expressions in question, as well as the last on. If the intersection of the resulting sets is empty, a potential word within the intersection can only be the empty word, which only is the case whenever all regular expressions are either ε or outermost nested by a Kleene-star. The following function $\mathsf{sol}_{\mathrm{pref}} : \mathtt{RegExC}_A \to 2^A$ for a regular expression R defined by

$$\mathsf{sol}_{\mathrm{pref}}(R) = \begin{cases} \varnothing & \text{if } R \in \{\,\varnothing, \varepsilon\,\}, \\ \{\,R\,\} & \text{if } R \in A, \\ \mathsf{sol}_{\mathrm{pref}}(R_1) & \text{if } R = R_1 \cdot R_2, \\ \mathsf{sol}_{\mathrm{pref}}(R_1) \cup \mathsf{sol}_{\mathrm{pref}}(R_2) & \text{if } R = R_1 \cup R_2, \\ \mathsf{sol}_{\mathrm{pref}}(R_1) & \text{if } R = R_1^*, \\ A \backslash \mathsf{sol}_{\mathrm{pref}}(R_1) & \text{if } R = \overline{R_1} \end{cases}$$

calculates the set of potential letters for the prefix of length 1 for any solution. A function for calculating the last letter of a potential solution can be calculated analogously. The predicate $\texttt{eReg} \subseteq \texttt{RegExC}_A$ is simply defined for a regular expression R by

$$\texttt{eReg}(R) \;\texttt{iff}\; \exists\, R_1 \in \texttt{RegExC}_A \,.\, R = R_1^* \vee R = \varepsilon.$$

This allows us to quickly determine whether all regular expressions of choice accept the empty word.

Again, this potentially allows avoiding the construction of the intersection automaton. Moreover, by using this construction we are able to assert additional formulae to the core solver leaking information about the potential solution of our product automaton.

Levering Length Information

Z3STR3 needs to be capable to reason about more than just the decidable theory of regular membership constraints, linear length constraints, and concatenation. Therefore we cannot directly pick any solution from a set of automata describing all solutions to a variable but try different length within an automaton. Within the implementation we involve Z3's arithmetic solver: We ask for an integer assignment which is consistent with our automata and afterwards try to construct a solution based on the gathered length. If a solution proves invalid we assert a conflict clause blocking this assignment within the integer solver and ask for a new solution. This arithmetic solver integration is present in the current mainline version of Z3STR3.

6.4.2 Detailed results of the evaluation

Within our evaluation we first want to leverage the performance gains Z3STR3RE gets by using our heuristics. Afterwards we compare our implementation on the previously described set of benchmarks against other solver.

In Figure 6.8 and Table 6.7 we summarise the performance of our heuristics. Once, we ran our solver disabling all heuristics and additionally partially enable certain heuristics. For the solver we refer to as Z3STR3RE

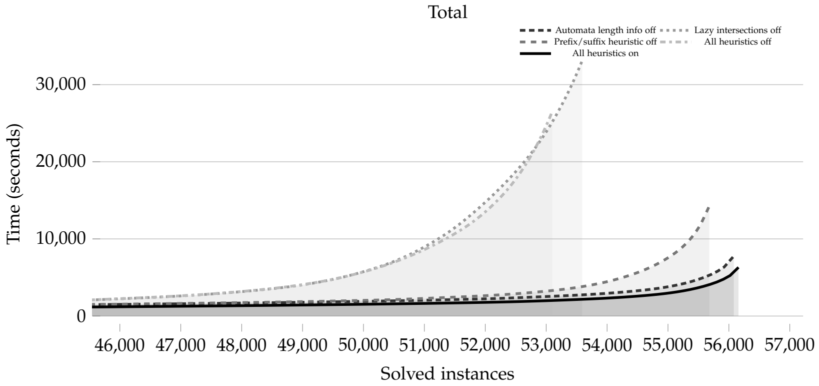


Figure 6.8. Cactus plot summarising performance of our heuristics on all benchmarks.

we enable all heuristics. If we disable all heuristics our solver gets 3.46x slower and solves 3057 less instances compared to all heuristics enabled. The automata length heuristic allows us solving only a couple more instances, namely 77, and adds a minor speed-up: Z3str3RE is 13.0% faster by using this heuristic. We achieve a boost of solved instances by activating the prefix-/suffix heuristic. We solve 479 additional instances and gain a speed-up overall of 75.0%. The most helpful heuristic is clearly the lazy intersection of automata. Z3str3RE is 3.32x faster by using this heuristic and solves 2567 additional instances. Overall these results demonstrate the usefulness of our heuristics, both by solving more instances and total solver runtime. Moreover, all heuristics can be used simultaneously for maximum efficacy.

Table 6.7. Comparison of our heuristics on all benchmarks.

	Automata length info off	Lazy intersections off	Prefix/suffix heuristic off	All heuristics off	All heuristics on
sat	33815	31509	33816	31013	33820
unsat	22266	22082	21863	22088	22338
unknown	285	323	287	315	291
timeout	890	3342	1290	3840	807
soundness error	0	0	0	0	0
program crashes	1	39	0	44	0
Total correct	56081	53591	55679	53101	56158
Time (s)	26475.22	101205.55	40934.03	104484.19	23419.8
Time w/o timeouts (s)	8675.22	34365.55	15134.03	27684.19	7279.8

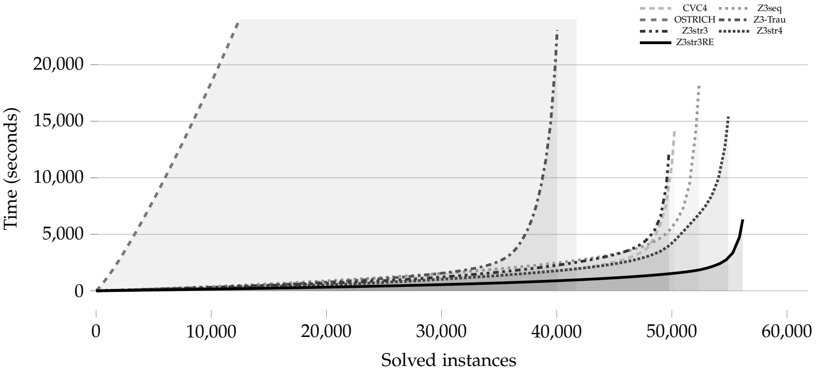


Figure 6.9. Cactus plot summarising performance on all benchmarks.

Comparing Z3str3RE against other solvers demonstrates the value of our approach. Our tool has the best performance on the benchmarks with respect to correctly classified instances and total time. The successor Z3str4 solves 1251 less instances correctly and is 1.59x times slower than Z3str3RE. It is worth mentioning again at this point, that Z3str4 uses the same algorithm for solving regular expression membership predicates as our tool does. Their arm selection procedure produces the overhead which leads to fewer solved cases within our selected timeout of 20 seconds. Secondly, within their arm for solving regular expression membership predicates they use Z3seq as a preprocessor. The sequence solver itself fails on solving 3804 compared to our solver. Moreover, it is 3.89x slower on our selected benchmarks. Z3str3RE is 5.6x faster than CVC4 solving a total of 5921 instances more. Comparing against Z3str3 which as mentioned

Table 6.8. Results summarising performance on all benchmarks.

	CVC4	Z3seq	OSTRICH	Z3-Trau	Z3str3	Z3str4	Z3str3RE
sat	28549	30726	22468	24184	30744	32664	33820
unsat	21688	21730	19284	21043	19162	22297	22338
unknown	5	6	10902	6517	588	38	291
timeout	7014	4794	4602	5512	6762	2257	807
soundness error	0	102	28	5329	166	54	0
program crashes	0	0	0	2485	0	0	0
Total correct	50237	52354	41724	39898	49740	54907	56158
Time (s)	154568.2	114437.0	297355.68	148134.88	147905.67	60627.72	23419.8
Time w/o timeouts (s)	14288.13	18557.0	205315.68	37894.88	12665.67	15487.72	7279.8

before uses an earlier version of our algorithm, Z3str3RE solves 6418 more instances within a total time which is 5.32x slower. Z3-Trau's performance on the selected benchmarks is slightly worth: the solver is 5.33x slower compared to our implementation and solves 16260 fewer cases correctly. Due to its Java implementation OSTRICH has the highest overall time. Our algorithm is 1170.0% faster than their implementation, solving 14434 more cases. A larger timeout might have gotten them some extra instance, but the solving time would suffer even more. Notably, using our validation setup we discovered multiple soundness issues within the competing solvers. Only CVC4 and our implementation prove to be sound. Detailed results on each benchmark set can be found in Appendix B.2.

It is worth mentioning, that these results differ compared to our work in [24] since we used other versions of the competing solvers.

6.4.3 Conclusion

We empirically showcased the power of our algorithm for regular expression membership constraints and its implementation in Z3str3RE via an extensive empirical comparison against six other state-of-the-art solvers over a large and diverse benchmark of more than 56993 instances. Over this entire benchmark suite, we show that Z3str3RE has the best overall performance. The presented method is very general and has wide applicability in the broad context of string solving. The challenges of using automata-based methods is addressed via prudent use of our heuristics in order to avoid performing expensive automata operations when possible. Our solver takes advantage of the compactness of automata in representing regular expressions, while at the same time mitigating the effects of expensive automata operations such as intersection by levering length information and lazy heuristics.

Chapter 7

Conclusion

"It's over and that's it. [...] I gave what I could."

Warhaus

In this thesis we went through different techniques of solving string constraints. We discovered an approach which is purely based on carefully encoding each required step into a propositional logic formula. Afterwards, we elaborated an approach based on a transition system by combining theoretical knowledge discovered decades ago with nowadays mechanisms to efficiently cope with real-world input. Continuing this trail, we use the ideas discovered while proving the decidability for certain theories involving regular membership constraints to build an efficient decision procedure for the aforementioned theory. To evaluate our approaches, we not only introduced a framework but also provide sophisticated test data collected from related literature. Furthermore, we analyse the test data for weaknesses and propose ways of optimising them. By using our infrastructure for testing and analysing string solvers, we showcase the performance of our approaches which are – even after being published some years ago – still perform equally or better then nowadays state of the art string solvers.

In the future we plan to extend all of the presented techniques in many ways: our first two proposals to solving string constraints are not able to reason about regular membership constraints. Therefore, we plan to integrate a similar method as we did within Z3str3RE into the propositional logic approach and also the transition system based technique. Initial experiments show that our automata based approaches we used for the

SAT approach only needs a few modifications to cope with regular membership constraints. Also the second way raises hope: modifying regular membership constraints in a similar to the linear length constraints seems to be feasible. We elaborated several heuristics optimising the algorithm implemented within Z3str3RE. In the future we plan to extend these heuristics to further speed up the solver. On the theoretical side our goal is to continue analysing real-world data to identify relevant sub theories, always hoping to find decidable fragments again revealing interesting algorithmic ideas to target industrial cases more efficiently.

Appendix A

Further Research

During my time in Kiel I was involved in several other but closely related activities. To keep this thesis streamlined we omitted these topics in the earlier chapters. Within this chapter we briefly introduce these results.

A.1 More on String Solving

We developed Z3str4, a string SMT solver for the quantifier free first order theory of strings, length constraints and regular membership predicates. Z3str4 has three core features: first, a novel length-abstraction algorithm that performs various string-length based abstractions and refinements along with a bit-vector backend; second, an arrangement-based solver with a bit-vector backend; third, an algorithm selection and constraint-sharing architecture which leverages the above-mentioned solvers along with the Z3 sequence solver. A paper describing all details was accepted at FM 2021 [91].

A.2 Prefix Normal Words

Prefix normal words are binary words in which each prefix has at least the same number of 1s as any factor of the same length. Firstly introduced in 2011, the problem of determining the index (amount of equivalence classes for a given word length) of the prefix normal equivalence relation is still open. We investigated two aspects of the problem, namely prefix normal palindromes and so-called collapsing words (extending the notion of critical words). We proved characterizations for both the palindromes and the collapsing words and show their connection in [51].

Furthermore, we investigate a generalisation for finite words over arbitrary finite alphabets, namely weighted prefix normality. We prove that weighted prefix normality is more expressive than binary prefix normality. Moreover, we investigate the existence of a weighted prefix normal form, since weighted prefix normality comes with several new peculiarities that did not already occur in the binary case. We characterise these issues and finally present a standard technique to obtain a generalised prefix normal form for all words over arbitrary, finite alphabets in [48].

A.3 Source Code Analysis

We developed a bridge between source code and the engine of UPPAAL which is massively expanding the realm of more traditional model checking technologies to include strategy synthesis algorithms — an aspect becoming more and more needed as software becomes increasingly parallel. Therefore, instead of reimplementing all these advances, we aim for using the UPPAAL ecosystem. Our approach uses the widespread intermediate language LLVM making UPPAAL readily available to the analysis of source code [75].

A.4 Participations in SAT and SMT solver competitions

In 2020 we participated in the SAT-COMP [14] on the AWS cloud track. We used a slightly modified version of TOPOSAT2 [47] developed by Ehlers. It is built on top of Glucose 3.0, but uses a bug-fixed version of the lockless clause sharing mechanism from ManySAT [59] for communication on one computation node rather than the lock-based implementation from Glucose Syrup. The communication between nodes uses MPI. Using this setup, we won a silver medal.

In 2020 and 2021 we entered SMT-COMP [17] using Z3STR4 (see Section A.1) on the quantifier free logic of strings (`QF_S`) and the quantifier free logic of strings and linear arithmetic (`QF_SLIA`). In 2020 we finished second, but only CVC4 entered on the aforementioned theories. During the time of the submission of this thesis the results for 2021 are not publicly available.

A.5 Teaching at Kiel University

During my time in Kiel I assisted two undergraduate courses, namely a course on theory of computation and a course on logic in computer science. These courses form the basis for teaching formal methods in Kiel. Over the last years, new study programs (computer science teacher training, business information systems, computer science for international students on master level) have been established, calling for changes to the courses. Guided by the experience gathered over time, course syllabi as well as teaching and examination formats and practices were adapted, resulting in a complex scheme. We share our experiences in [52] and review this development, with particular focus on managing heterogeneity and bridging the gap between actual and required qualification of enrolling students. Key ingredients of our teaching methods are a spiral approach, frequent testing, supervised learning time, and a game.

In the beginning of 2020 when COVID-19 arose, after a short moment of shock our university decided that the students have to be able to pursue their studies for guaranteeing a degree in the expected time since most of them faced immediate financial problems due to the loss of their student jobs. This implied, for us as teachers, that we had to reorganise not only the teaching methods from nearly one day to the next, but we also had to come up with an adjusted way of examinations which had to take place in person with pen and paper under strict hygiene rules. On the other hand the correction should avoid personal contacts. We developed a framework which allowed us to correct the digitalised exams safely at home while providing the high standards given by the general data protection regulation of our country. Moreover, the time spent in the offices could be reduced to a minimum thanks to automatically generated exam sheets, automatically re-digitalised and sorted worked-on exams. A description of this frame work is available in [50].

Appendix B

Detailed results of the empirical evaluation

In this part of the appendix we present the performance of our approaches on the specific benchmark sets, to show how diverse instances influence the individual performance of our solvers.

Table B.1. Results for Woorpje Track 1.

	CVC4	Z3seq	Z3str3	Z3str4	Z3-Trau	OSTRICH	woorpjeSAT
sat	200	196	192	196	196	18	200
unsat	0	0	0	0	1	0	0
unknown	0	0	0	0	0	182	0
timeout	0	4	8	4	3	0	0
soundness error	0	0	0	0	9	0	0
program crashes	0	0	0	0	0	0	0
Total correct	200	196	192	196	188	18	200
Time (s)	5.87	125.15	170.96	86.67	112.55	281.14	17.33
Time w/o timeouts (s)	5.87	45.15	10.96	6.67	52.55	281.14	17.33

Table B.2. Results for Woorpje Track 2.

	CVC4	Z3seq	Z3str3	Z3str4	Z3-Trau	OSTRICH	woorpjeSAT
sat	4	4	0	4	3	0	5
unsat	0	0	0	0	3	0	0
unknown	0	0	9	0	0	9	0
timeout	5	5	0	5	3	0	4
soundness error	0	0	0	0	2	0	0
program crashes	0	0	0	0	0	0	0
Total correct	4	4	0	4	4	0	5
Time (s)	114.07	108.88	0.47	122.41	86.82	7.07	83.98
Time w/o timeouts (s)	14.07	8.88	0.47	22.41	26.82	7.07	3.98

Table B.3. Results for Woorpje Track 3.

	CVC4	Z3seq	Z3str3	Z3str4	Z3-Trau	OSTRICH	woorpjeSAT
sat	134	128	83	92	134	1	147
unsat	44	44	43	44	46	3	42
unknown	0	0	2	0	0	196	2
timeout	22	28	72	64	20	0	9
soundness error	0	0	0	0	6	0	0
program crashes	0	0	0	0	0	0	2
Total correct	178	172	126	136	174	4	189
Time (s)	519.08	649.44	1466.46	1298.57	681.24	303.77	198.59
Time w/o timeouts (s)	79.08	89.44	26.46	18.57	281.24	303.77	18.59

Table B.4. Results for Woorpje Track 4.

	CVC4	Z3seq	Z3str3	Z3str4	Z3-Trau	OSTRICH	woorpjeSAT
sat	103	101	76	94	77	14	104
unsat	96	96	104	96	95	2	93
unknown	0	0	1	0	0	182	1
timeout	1	3	19	10	28	2	2
soundness error	0	0	8	0	5	0	0
program crashes	0	0	0	0	0	0	1
Total correct	199	197	172	190	167	16	197
Time (s)	87.52	85.79	439.23	219.9	592.94	1398.97	128.22
Time w/o timeouts (s)	67.52	25.79	59.23	19.9	32.94	1358.97	88.22

Table B.5. Results for Woorpje Track 5.

	CVC4	Z3seq	Z3str3	Z3str4	Z3-Trau	OSTRICH	woorpjeSAT
sat	176	175	168	176	150	1	176
unsat	24	24	24	24	43	0	14
unknown	0	0	0	0	0	199	0
timeout	0	1	8	0	7	0	10
soundness error	0	0	0	0	29	0	0
program crashes	0	0	0	0	0	0	0
Total correct	200	199	192	200	164	1	190
Time (s)	7.34	31.32	170.9	8.55	311.84	411.16	210.54
Time w/o timeouts (s)	7.34	11.32	10.9	8.55	171.84	411.16	10.54

B.1 Evaluation of the SAT approach

We visualise WoorpjeBench Track 1 in Table B.1, respectively Figure B.1. woorpjeSAT solves as many instances as CVC4 (the best solver on this track), but is 1.95x slower.

We visualise WoorpjeBench Track 2 in Table B.2, respectively Figure B.2. woorpjeSAT solves more instances than any other solver and is 25% faster compared to the second best solver CVC4.

We visualise WoorpjeBench Track 3 in Table B.3, respectively Figure B.3. woorpjeSAT is 1.61x faster than CVC4, the second best solver on this track, and solves 11 more instances than CVC4.

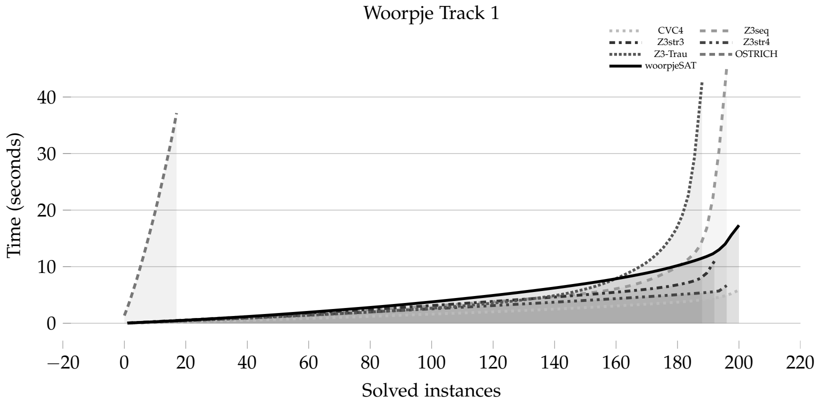


Figure B.1. Cactus plot summarising performance on Woorpje Track 1.

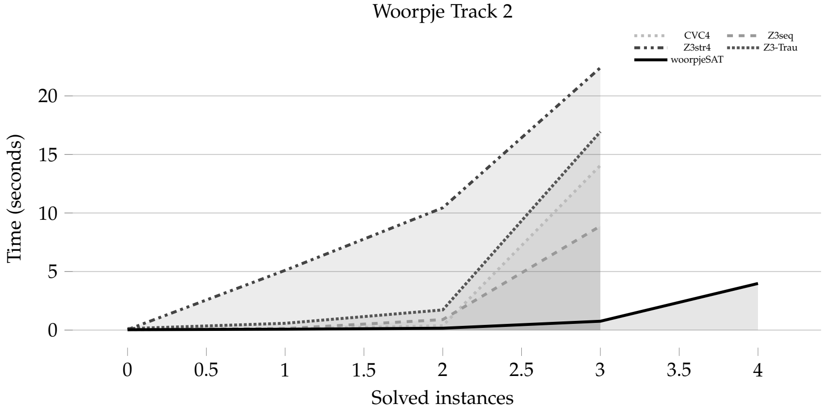


Figure B.2. Cactus plot summarising performance on Woorpje Track 2.

We visualise WoorpjeBench Track 4 in Table B.4, respectively Figure B.4. woorpjeSAT fails on solving 2 instances CVC4 does (the best solver on this track) and is 0.47x slower on this track.

We visualise WoorpjeBench Track 5 in Table B.5, respectively Figure B.5. woorpjeSAT fails on solving 10 instances CVC4 does (the best

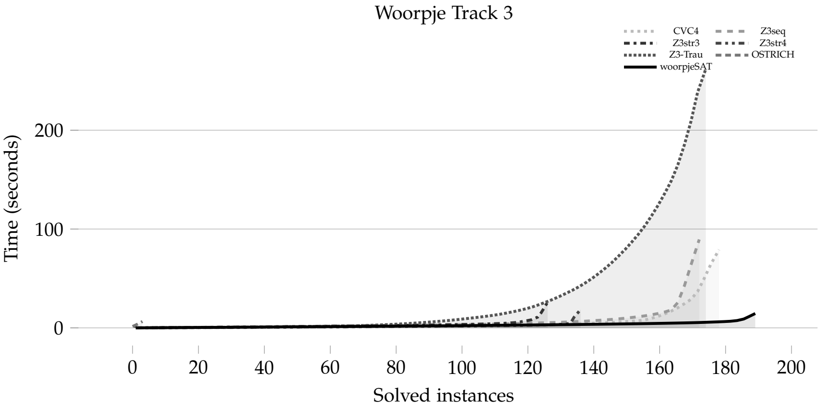


Figure B.3. Cactus plot summarising performance on Woorpje Track 3.

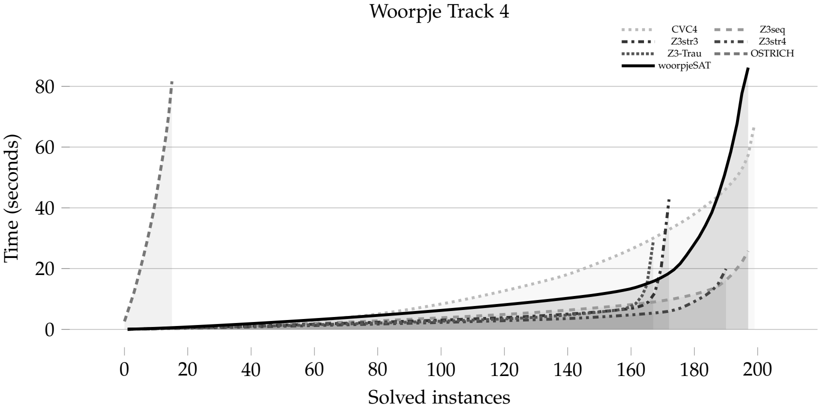


Figure B.4. Cactus plot summarising performance on Woorpje Track 4.

solver on this track) and is 27.68x slower on this track. This drawback is caused by the naive implementation of the linear constraint handling.

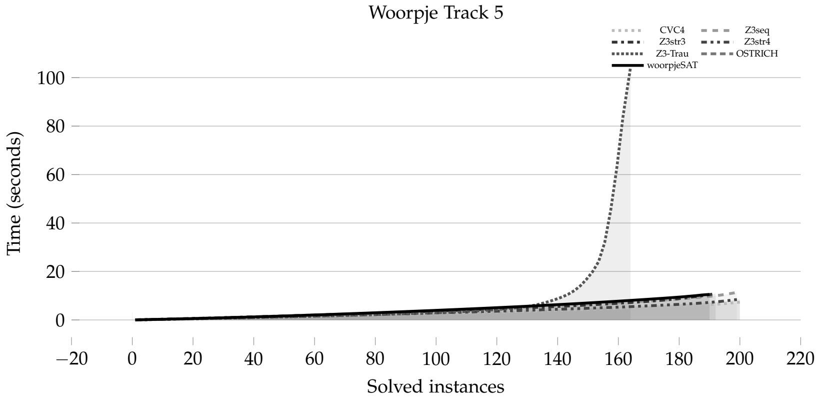


Figure B.5. Cactus plot summarising performance on Woorpje Track 5.

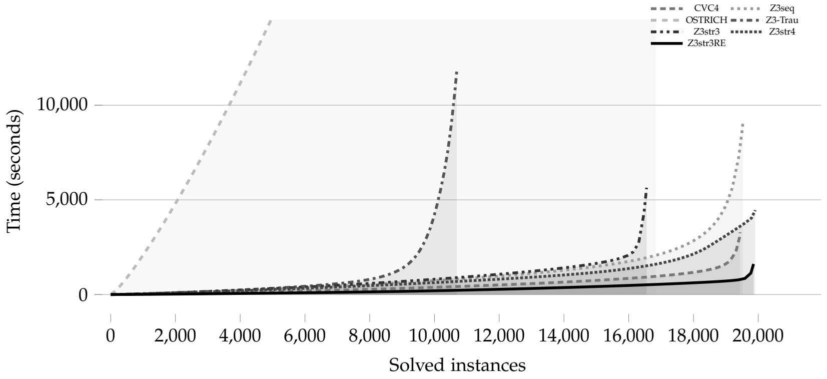


Figure B.6. Cactus plot summarising performance on AutomatArk.

B.2 Evaluation of the procedure for solving regular membership constraints

We visualise AutomatArk in Figure B.6, respectively Figure B.6. Z3str3RE solves 45 instances less than Z3str4, but is 0.46x faster.

We visualise StringFuzz-regex-generated in Table B.7, respectively

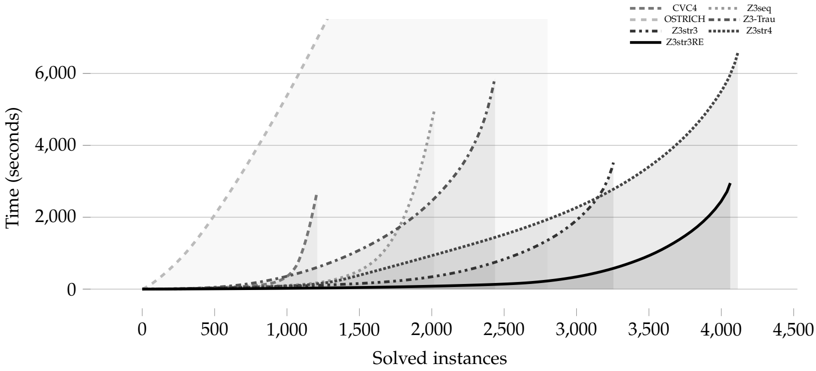


Figure B.7. Cactus plot summarising performance on Stringfuzz RegEx Generated.

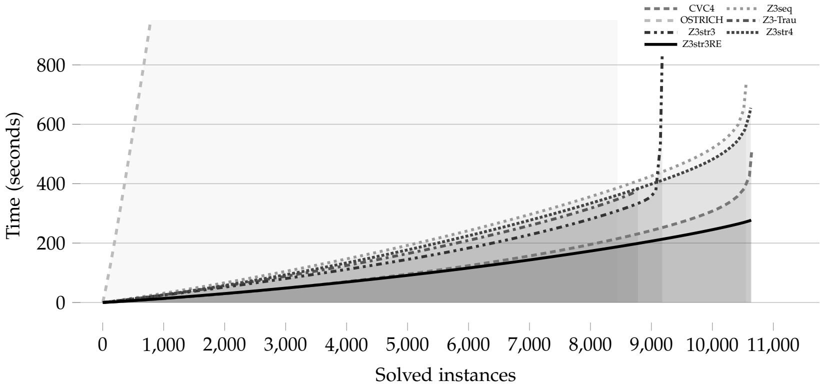


Figure B.8. Cactus plot summarising performance on Stringfuzz RegEx Transformed.

Figure B.7. Z3str3RE solves 52 instances less than Z3str4, but is 0.49x faster.

We visualise StringFuzz-regex-transformed in Table B.8, respectively Figure B.8. Z3str3RE solves 7 more instances than Z3str4, but is 0.5x slower.

We visualise Regex-Collected in Table B.9, respectively Figure B.9.

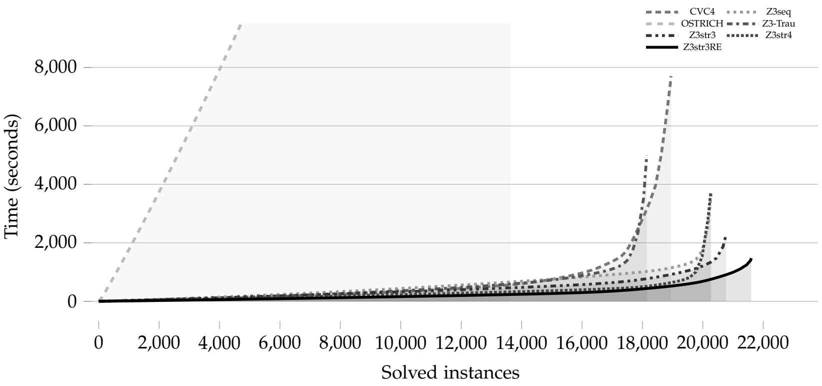


Figure B.9. Cactus plot summarising performance on RegEx Collected.

Table B.6. Results for AutomatArk.

	CVC4	Z3seq	OSTRICH	Z3-Trau	Z3str3	Z3str4	Z3str3RE
sat	14157	14261	11458	8172	12155	14457	14436
unsat	5282	5280	5382	3822	4407	5448	5421
unknown	5	6	16	5058	315	1	0
timeout	535	432	3123	2927	3102	73	122
soundness error	0	17	0	1304	11	3	0
program crashes	0	0	0	1071	0	0	0
Total correct	19439	19524	16840	10690	16551	19902	19857
Time (s)	14065.03	17747.98	153148.85	79156.71	67773.38	5921.77	4054.42
Time w/o timeouts (s)	3365.03	9107.98	90688.85	20616.71	5733.38	4461.77	1614.42

Table B.7. Results for Stringfuzz RegEx Generated.

	CVC4	Z3seq	OSTRICH	Z3-Trau	Z3str3	Z3str4	Z3str3RE
sat	768	1319	1979	1621	3285	3284	3232
unsat	441	697	821	824	32	830	830
unknown	0	0	1	192	0	6	0
timeout	2961	2154	1369	1533	853	50	108
soundness error	0	0	0	8	62	0	0
program crashes	0	0	0	192	0	0	0
Total correct	1209	2016	2800	2437	3255	4114	4062
Time (s)	61932.71	48019.13	52097.02	36600.02	20797.26	7597.63	5109.0
Time w/o timeouts (s)	2712.64	4939.13	24717.02	5940.02	3737.26	6597.63	2949.0

Z3str3RE solves 1341 more instances than Z3str4 and is 2.52x faster.

Table B.8. Results for Stringfuzz RegEx Transformed.

	CVC4	Z3seq	OSTRICH	Z3-Trau	Z3str3	Z3str4	Z3str3RE
sat	4625	4613	3899	3672	4334	4617	4599
unsat	6018	6005	4549	6282	4909	6062	6037
unknown	0	0	2233	721	1	0	6
timeout	39	64	1	7	1438	3	40
soundness error	0	82	5	1241	82	50	0
program crashes	0	0	0	718	0	0	0
Total correct	10643	10536	8443	8713	9161	10629	10636
Time (s)	1285.61	2042.52	20972.86	706.17	29601.24	719.72	1077.46
Time w/o timeouts (s)	505.61	762.52	20952.86	566.17	841.24	659.72	277.46

Table B.9. Results for RegEx Collected.

	CVC4	Z3seq	OSTRICH	Z3-Trau	Z3str3	Z3str4	Z3str3RE
sat	8999	10533	5132	10719	10970	10306	11553
unsat	9947	9748	8532	10115	9814	9957	10050
unknown	0	0	8652	546	272	31	285
timeout	3479	2144	109	1045	1369	2131	537
soundness error	0	3	23	2776	11	1	0
program crashes	0	0	0	504	0	0	0
Total correct	18946	20278	13641	18058	20773	20262	21603
Time (s)	77284.84	46627.37	71136.95	31671.98	29733.79	46388.6	13178.92
Time w/o timeouts (s)	7704.84	3747.37	68956.95	10771.98	2353.79	3768.6	2438.92

Bibliography

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Julian Dolby, Petr Janků, Hsin-Hung Lin, Lukáš Holík, and Wei-Cheng Wu. "Efficient handling of string-number conversion". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2020, pp. 943–957.

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukáš Holik, Ahmed Rezine, and Philipp Rümmer. "Trau: SMT solver for string constraints". In: *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE. 2018, pp. 1–5.

[3] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukáš Holík, Ahmed Rezine, and Philipp Rümmer. "Flatten and conquer: a framework for efficient analysis of string constraints". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: ACM, 2017, pp. 602–617.

[4] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. "Norn: an smt solver for string constraints". In: *International conference on Computer Aided Verification*. Springer. 2015, pp. 462–469.

[5] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. "String constraints for verification". In: *International Conference on Computer Aided Verification*. Springer. 2014, pp. 150–166.

[6] Ignasi Abío and Peter J. Stuckey. "Encoding linear constraints into SAT". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2014, pp. 75–91.

[7] Dan Allen and Sarah White. *Asciidoctor*. `https://asciidoctor.org`. Accessed: 2021-01-06.

[8] Roberto Amadini. *A survey on string constraint solving*. 2020. arXiv: `2002.02376 [cs.AI]`.

[9] Roberto Amadini, Graeme Gange, and Peter Stuckey. "Sweep-based propagation for string constraint solving". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1. 2018.

[10] Gilles Audemard and Laurent Simon. "On the glucose sat solver". In: *International Journal on Artificial Intelligence Tools* 27.01 (2018), p. 1840001.

[11] Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. "Automata-based model counting for string constraints". In: *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Cham: Springer International Publishing, 2015, pp. 255–272.

[12] John Backes, Pauline Bolignano, Byron Cook, Andrew Gacek, Kasper Soe Luckow, Neha Rungta, Martin Schaef, Cole Schlesinger, Rima Tanash, Carsten Varming, et al. "One-click formal methods". In: *IEEE Software* 36.6 (2019), pp. 61–65.

[13] Thomas Ball and Jakub Daniel. "Deconstructing dynamic symbolic execution". In: *Proc. Marktoberdorf Summer School on Dependable Software Systems Engineering* (2014).

[14] Tomáš Balyo, Nils Froleyks, Marijn J.H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda. *Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions*. University of Helsinki, Department of Computer Science, 2020.

[15] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. "Saner: composing static and dynamic analysis to validate sanitization in web applications". In: *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE. 2008, pp. 387–401.

[16] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. "CVC4". In: *International Conference on Computer Aided Verification*. Springer. 2011, pp. 171–177.

[17] Clark Barrett, Leonardo De Moura, and Aaron Stump. "SMT-COMP: Satisfiability modulo theories competition". In: *International Conference on Computer Aided Verification*. Springer. 2005, pp. 20–23.

[18] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. `www.SMT-LIB.org`. 2016.

[19] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. "The SMT-LIB standard: Version 2.0". In: *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*. Vol. 13. 2010, p. 14.

[20] Clark Barrett and Cesare Tinelli. "Satisfiability modulo theories". In: *Handbook of Model Checking*. Springer, 2018, pp. 305–343.

[21] Murphy Berzish. "Z3str4: a solver for theories over strings". In: (2021).

[22] Murphy Berzish, Joel D Day, Vijay Ganesh, Mitja Kulczynski, Florin Manea, Federico Mora, and Dirk Nowotka. "String theories involving regular membership predicates: from practice to theory and back". In: *International Conference on Combinatorics on Words*. Springer. 2021.

[23] Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. "Z3str3: A string solver with theory-aware heuristics". In: *2017 Formal Methods in Computer Aided Design (FMCAD)*. IEEE. 2017, pp. 55–59.

[24] Murphy Berzish, Mitja Kulczynski, Federico Mora, Florin Manea, Joel D. Day, Dirk Nowotka, and Vijay Ganesh. "An SMT Solver for Regular Expressions and Linear Arithmetic over String Length". In: *Computer Aided Verification*. Springer, 2021, pp. 289–312.

[25] Murphy Berzish, Mitja Kulczynski, Federico Mora, Dirk Nowotka, and Vijay Ganesh. *Z3str4 website*. `http://z3str4.github.io`. Accessed: 2021-06-11.

[26] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. "Path feasibility analysis for string-manipulating programs". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2009, pp. 307–321.

[27] Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. "Stringfuzz: a fuzzer for string solvers". In: *International Conference on Computer Aided Verification*. Springer. 2018, pp. 45–51.

[28] Tegan Brennan, Nestan Tsiskaridze, Nicolás Rosner, Abdulbaki Aydin, and Tevfik Bultan. "Constraint normalization and parameterized caching for quantitative program analysis". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 2017, pp. 535–546.

[29] Tevfik Bultan, Fang Yu, Muath Alkhalaf, and Abdulbaki Aydin. *String analysis for software verification and security*. Springer, 2017.

[30] Taolue Chen, Matthew Hague, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. "Decision procedures for path feasibility of string-manipulating programs with complex operations". In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), p. 49.

[31] Ting Chen, Xiao-song Zhang, Rui-dong Chen, Bo Yang, and Yang Bai. "Conpy: Concolic Execution Engine for Python Applications". In: *International Conference on Algorithms and Architectures for Parallel Processing*. Springer. 2014, pp. 150–163.

[32] Christian Choffrut and Juhani Karhumäki. "Combinatorics of words". In: *Handbook of formal languages*. Springer, 1997, pp. 329–438.

[33] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. "Precise analysis of string expressions". In: *International Static Analysis Symposium*. Springer. 2003, pp. 1–18.

[34] Marek Chrobak. "Finite automata and unary languages". In: *Theor. Comput. Sci.* 47.3 (1986), pp. 149–158.

[35] Geoffrey Chu, Peter J Stuckey, Andreas Schutt, Thorsten Ehlers, Graeme Gange, and Kathryn Francis. "Chuffed, a lazy clause generation solver". In: *URL: https://github. com/chuffed/chuffed* (2018).

[36] Vasek Chvatal. *Linear programming*. W. H. Freeman and Company, 1983.

[37] CVE. *Common vulnerabilities and exposures*. `http://www.cve.mitre.org`. Accessed: 2021-03-01.

[38] Loris D'Antoni and Margus Veanes. "The power of symbolic automata and transducers". In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*. Ed. by Rupak Majumdar and Viktor Kuncak. Vol. 10426. Lecture Notes in Computer Science. Springer, 2017, pp. 47–67.

[39] Martin Davis, George Logemann, and Donald Loveland. "A machine program for theorem-proving". In: *Communications of the ACM* 5.7 (1962), pp. 394–397.

[40] Joel D. Day, Thorsten Ehlers, Mitja Kulczynski, Florin Manea, Dirk Nowotka, and Danny Bøgsted Poulsen. "On solving word equations using SAT". In: *International Conference on Reachability Problems*. Springer. 2019, pp. 93–106.

[41] Joel D. Day, Vijay Ganesh, Paul He, Florin Manea, and Dirk Nowotka. "The satisfiability of word equations: decidable and undecidable theories". In: *Reachability Problems - 12th International Conference, RP 2018, Proceedings*. Ed. by Igor Potapov and Pierre-Alain Reynier. Vol. 11123. Springer, 2018, pp. 15–29.

[42] Joel D. Day, Mitja Kulczynski, Florin Manea, Dirk Nowotka, and Danny Bøgsted Poulsen. "Rule-based word equation solving". In: *Proceedings of the 8th International Conference on Formal Methods in Software Engineering*. 2020, pp. 87–97.

[43] Joel D. Day, Florin Manea, and Dirk Nowotka. "The hardness of solving simple word equations". In: *Proc. MFCS 2017*. Vol. 83. LIPIcs. 2017, 18:1–18:14.

[44] Leonardo De Moura and Nikolaj Bjørner. "Satisfiability modulo theories: introduction and applications". In: *Communications of the ACM* 54.9 (2011), pp. 69–77.

[45] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *TACAS*. Springer. 2008, pp. 337–340.

[46] Heinz-Dieter Ebbinghaus and Jörg Flum. *Finite model theory*. Springer Science & Business Media, 2005.

[47] Thorsten Ehlers. "SAT and CP-Parallelisation and Applications". PhD thesis. Christian-Albrechts Universität Kiel, 2017.

[48] Yannik Eikmeier, Pamela Fleischmann, Mitja Kulczynski, and Dirk Nowotka. "Weighted Prefix Normal Words: Mind the Gap". In: *Developments in Language Theory, Proceedings*. Springer. 2021.

[49] Michael Emmi, Rupak Majumdar, and Koushik Sen. "Dynamic test input generation for database applications". In: *Proceedings of the 2007 international symposium on Software testing and analysis*. 2007, pp. 151–162.

[50] Pamela Fleischmann, Mitja Kulczynski, and Dirk Nowotka. *The show must go on – examination during a pandemic*. 2021. arXiv: `2107.04014 [cs.CY]`.

[51] Pamela Fleischmann, Mitja Kulczynski, Dirk Nowotka, and Danny Bøgsted Poulsen. "On collapsing prefix normal words". In: *International Conference on Language and Automata Theory and Applications*. Springer. 2020, pp. 412–424.

[52] Pamela Fleischmann, Mitja Kulczynski, Dirk Nowotka, and Thomas Wilke. "Managing heterogeneity and bridging the gap in teaching formal methods". In: *Formal Methods Teaching Workshop*. Springer. 2019, pp. 181–195.

[53] Thom Frühwirth and Slim Abdennadher. *Essentials of constraint programming*. Springer Science & Business Media, 2003.

[54] Vijay Ganesh, Sergey Berezin, and David L. Dill. "Deciding Presburger Arithmetic by Model Checking and Comparisons with Other Methods". In: *Formal Methods in Computer-Aided Design, 4th International Conference, FMCAD 2002, Portland, OR, USA, November 6-8, 2002, Proceedings*. 2002, pp. 171–186.

[55] Pawel Gawrychowski. "Chrobak normal form revisited, with applications". In: *Implementation and Application of Automata - 16th International Conference, CIAA 2011, Blois, France, July 13-16, 2011. Proceedings*. 2011, pp. 142–153.

[56] Carl Gould, Zhendong Su, and Premkumar Devanbu. "Static checking of dynamically generated queries in database applications". In: *Proceedings. 26th International Conference on Software Engineering*. IEEE. 2004, pp. 645–654.

[57] Hagenah, Christian and Muscholl, Anca. "Computing e nfa from regular expressions in o(n log2(n)) time". In: *RAIRO-Theor. Inf. Appl.* 34.4 (2000), pp. 257–277.

[58] Matthew Hague. "Strings at MOSCA". In: *ACM SIGLOG News* 6.4 (2019), pp. 4–22.

[59] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. "ManySAT: a parallel SAT solver". In: *Journal on Satisfiability, Boolean Modeling and Computation* 6.4 (2010), pp. 245–262.

[60] Juris Hartmanis. "Computers and intractability: a guide to the theory of np-completeness (michael r. garey and david s. johnson)". In: *Siam Review* 24.1 (1982), p. 90.

[61] David Hilbert. "Mathematische Probleme". In: *Nachrichten von der Gesellschaft der Wissenschaften zu Göttingen, Mathematisch-Physikalische Klasse* 1900 (1900), pp. 253–297.

[62] Lukáš Holík, Petr Janků, Anthony W. Lin, Philipp Rümmer, and Tomáš Vojnar. "String constraints with concatenation and transducers solved efficiently". In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), p. 4.

[63] Michal Hospodár, Galina Jirásková, and Peter Mlynárcik. "A survey on fooling sets as effective tools for lower bounds on nondeterministic complexity". In: *Adventures Between Lower Bounds and Higher Altitudes - Essays Dedicated to Juraj Hromkovič on the Occasion of his 60th Birthday*. 2018, pp. 17–32.

[64] *IBM AppScan Source web site*. https://www.ibm.com/dk-da/security/application-security/appscan.

[65] Artur Jeż. "Recompression: a simple and powerful technique for word equations". In: *30th International Symposium on Theoretical Aspects of Computer Science, STACS 2013, February 27 - March 2, 2013, Kiel, Germany*. 2013, pp. 233–244.

[66] Artur Jeż. "Word equations in nondeterministic linear space". In: *Proc. ICALP 2017*. Vol. 80. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 95:1–95:13.

[67] David S. Johnson. "The NP-completeness column: an ongoing guide". In: *Journal of Algorithms* 6.3 (1985), pp. 434–451.

[68] Juhani Karhumäki, Filippo Mignosi, and Wojciech Plandowski. "The expressibility of languages and relations by word equations". In: *Journal of the ACM (JACM)* 47.3 (2000), pp. 483–505.

[69] Scott Kausler and Elena Sherman. "Evaluation of string constraint solvers in the context of symbolic execution". In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 2014, pp. 259–270.

[70] IUriı Ilich Khmelevskiı. *Equations in free semigroups*. 107. American Mathematical Soc., 1976.

[71] Adam Kiezun, Vijay Ganesh, Philip J Guo, Pieter Hooimeijer, and Michael D Ernst. "Hampi: a solver for string constraints". In: *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM. 2009, pp. 105–116.

[72] James C King. "Symbolic execution and program testing". In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[73] Fabrice Kordon and Francis Hulin-Hubard. "Benchkit, a tool for massive concurrent benchmarking". In: *2014 14th International Conference on Application of Concurrency to System Design*. 2014, pp. 159–165.

[74] Dexter Kozen. "Lower bounds for natural proof systems". In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. 1977, pp. 254–266.

[75] Mitja Kulczynski, Axel Legay, Dirk Nowotka, and Danny Bøgsted Poulsen. "Analysis of Source Code Using UPPAAL". In: *Electronic Proceedings in Theoretical Computer Science* 338 (2021), pp. 31–38.

[76] Mitja Kulczynski, Florin Manea, Dirk Nowotka, and Danny Bøgsted Poulsen. "The power of string solving: simplicity of comparison". In: *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*. 2020, pp. 85–88.

[77] Mitja Kulczynski, Florin Manea, Dirk Nowotka, and Danny Bøgsted Poulsen. "ZaligVinder: A generic test framework for string solvers". In: *Journal of Software: Evolution and Process* (2021), e2400.

[78] Frank W. Levi. "On semigroups". In: *Bull. Calcutta Math. Soc* 36.141-146 (1944), p. 82.

[79] Guodong Li and Indradeep Ghosh. "Pass: string solving with parameterized array and interval automaton". In: *Haifa Verification Conference*. Springer. 2013, pp. 15–31.

[80] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. "A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions". In: *Proceedings of the 26th International Conference on Computer Aided Verification*. CAV'14. 2014, pp. 646–662.

[81] Tianyi Liang, Andrew Reynolds, Nestan Tsiskaridze, Cesare Tinelli, Clark Barrett, and Morgan Deters. "An efficient SMT solver for string constraints". In: *Formal Methods in System Design* 48.3 (2016), pp. 206–234.

[82] Tianyi Liang, Nestan Tsiskaridze, Andrew Reynolds, Cesare Tinelli, and Clark Barrett. "A decision procedure for regular membership and length constraints over unbounded strings". In: *Frontiers of Combining Systems - 10th International Symposium, FroCoS 2015, Wroclaw, Poland, September 21-24, 2015. Proceedings*. 2015, pp. 135–150.

[83] Anthony W. Lin and Rupak Majumdar. "Quadratic Word Equations with Length Constraints, Counter Systems, and Presburger Arithmetic with Divisibility". In: *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*. Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 11138. Lecture Notes in Computer Science. Springer, 2018, pp. 352–369.

[84] Anthony Widjaja Lin and Pablo Barceló. "String solving with word equations and transducers: towards a logic for analysing mutation XSS". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by Rastislav Bodík and Rupak Majumdar. ACM, 2016, pp. 123–136.

[85] M. Lothaire. *Combinatorics on words*. Addison-Wesley, 1983.

[86] Loi Luu, Shweta Shinde, Prateek Saxena, and Brian Demsky. "A model counter for constraints over unbounded strings". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2014, pp. 565–576.

[87] Gennady S. Makanin. "The problem of solvability of equations in a free semigroup". In: *Sbornik: Mathematics* 32.2 (1977), pp. 129–198.

[88] Yuri Matiyasevich. "Word equations, fibonacci numbers, and hilbert's tenth problem". In: *Workshop on Fibonacci words*. Vol. 43. 2007, pp. 36–39.

[89] Yuri Vladimirovich Matiyasevich. "A connection between systems of words-and-lengths equations and Hilbert's tenth problem". In: *Zapiski Nauchnykh Seminarov POMI* 8 (1968), pp. 132–144.

[90] Yasuhiko Minamide. "Static approximation of dynamically generated web pages". In: *Proceedings of the 14th international conference on World Wide Web*. 2005, pp. 432–441.

[91] Federico Mora, Murphy Berzish, Mitja Kulczynski, Dirk Nowotka, and Vijay Ganesh. "Z3str4: a multi-armed string solver". In: *International Symposium on Formal Methods*. Springer. 2021, pp. 389–406.

[92] Hung Viet Nguyen, Christian Kästner, and Tien N Nguyen. "Building call graphs for embedded client-side code in dynamic web applications". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2014, pp. 518–529.

[93] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. "Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL (T)". In: *Journal of the ACM (JACM)* 53.6 (2006), pp. 937–977.

[94] Wojciech Plandowski. “Satisfiability of word equations with constants is in PSPACE”. In: *Foundations of Computer Science, 1999. 40th Annual Symposium on*. IEEE. 1999, pp. 495–500.

[95] Wojciech Plandowski and Wojciech Rytter. “Application of Lempel-Ziv encodings to the solution of word equations”. In: *International Colloquium on Automata, Languages, and Programming*. Springer. 1998, pp. 731–742.

[96] Gideon Redelinghuys, Willem Visser, and Jaco Geldenhuys. “Symbolic execution of programs with strings”. In: *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*. SAICSIT ’12. Pretoria, South Africa, 2012, pp. 139–148.

[97] Andrew Reynolds, Maverick Woo, Clark Barrett, David Brumley, Tianyi Liang, and Cesare Tinelli. “Scaling up DPLL (T) string solvers using context-dependent simplification”. In: *International Conference on Computer Aided Verification*. Springer. 2017, pp. 453–474.

[98] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[99] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. “A symbolic execution framework for JavaScript”. In: *2010 IEEE Symposium on Security and Privacy*. IEEE. 2010, pp. 513–528.

[100] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. *Kaluza web site*. `webblaze.cs.berkeley.edu/2010/kaluza/`. Accessed: 2020-01-01.

[101] Christian Schulte, Mikael Lagerkvist, and Guido Tack. “Gecode”. In: *Software download and online material at the website: http://www.gecode. org* (2006), pp. 11–13.

[102] Joseph Scott, Federico Mora, and Vijay Ganesh. “BanditFuzz: A Reinforcement-Learning based Performance Fuzzer for SMT Solvers”. In: (2020), pp. 68–86.

[103] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. “Jalangi: a selective record-replay and dynamic analysis framework for JavaScript”. In: *In Proc. ESEC/FSE*. ACM, 2013, pp. 488–498.

[104] Michael Sipser. *Introduction to the theory of computation*. Cengage learning, 2012.

[105] *SMT-LIB benchmarks Repository*. `https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks`. Accessed: 2021-06-13.

[106] Caleb Stanford, Margus Veanes, and Nikolaj Bjørner. *Symbolic boolean derivatives for efficiently solving extended regular expression constraints*. Tech. rep. MSR-TR-2020-25. Microsoft, 2020.

[107] Larry J. Stockmeyer. *The complexity of decision problems in automata theory and logic*. PhD. Thesis, MIT, 1974.

[108] Takaaki Tateishi, Marco Pistoia, and Omer Tripp. “Path-and index-sensitive string analysis based on monadic second-order logic”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22.4 (2013), pp. 1–33.

[109] Julian Thomé. *Smtlibv2-grammar*. `https://github.com/julianthome/smtlibv2-grammar`. Accessed: 2021-01-06.

[110] Julian Thome, Lwin Khin Shar, Domenico Bianculli, and Lionel Briand. “An integrated approach for effective injection vulnerability analysis of web applications through security slicing and hybrid constraint solving”. In: *IEEE Transactions on Software Engineering* 46.2 (2018), pp. 163–195.

[111] Cesare Tinelli, Clark Barrett, and Pascal Fontaine. *Smt: theory of strings*. `http://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml`. Accessed: 2021-01-06.

[112] Antoine Toullalan and Fabrice Kordon. “Mcc 2021”. In: *Communications of the ACM* (2021).

[113] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. “Model counting for recursively-defined strings”. In: *International Conference on Computer Aided Verification*. Springer. 2017, pp. 399–418.

[114] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. "S3: a symbolic string solver for vulnerability detection in web applications". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 2014, pp. 1232–1243.

[115] Pavol Voda. *The constraint language trilogy: semantics and computations*. Tech. rep. Complete Logic Systems, 1988.

[116] Hung-En Wang, Tzung-Lin Tsai, Chun-Han Lin, Fang Yu, and Jie-Hong R Jiang. "String analysis via automata manipulation with logic circuit representation". In: *International Conference on Computer Aided Verification*. Springer. 2016, pp. 241–260.

[117] Gary Wassermann and Zhendong Su. "Sound and precise analysis of web applications for injection vulnerabilities". In: *PLDI*. 2007, pp. 32–41.

[118] Philipp Wendler and Dirk Beyer. *Sosy-lab/benchexec: release 3.8*. Version 3.8. May 2021. URL: https://doi.org/10.5281/zenodo.4773326.

[119] XKCD. *A webcomic of romance, sarcasm, math, and language*. https://xkcd.com. Accessed: 2021-03-01.

[120] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. "STRANGER: An Automata-based String Analysis Tool for PHP". In: *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'10. Paphos, Cyprus: Springer-Verlag, 2010, pp. 154–157. ISBN: 3-642-12001-6, 978-3-642-12001-5.

[121] Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Murphy Berzish, Julian Dolby, and Xiangyu Zhang. "Z3str2: an efficient solver for strings, regular expressions, and length constraints". In: *Formal Methods in System Design* (2016), pp. 1–40.

[122] Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Julian Dolby, and Xiangyu Zhang. "Effective search-space pruning for solvers of string equations, regular expressions and length constraints". In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. 2015, pp. 235–254.

[123] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. “Z3-str: a z3-based string solver for web application analysis”. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 2013, pp. 114–124.