BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

HEURISTICS IN STRING SOLVING

HEURISTIKY VE STRING SOLVINGU

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. MICHAL ŘEZÁČ**
AUTOR PRÁCE

SUPERVISOR **Mgr. JURAJ SÍČ**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

153659

Institut:	Department of Intelligent Systems (DITS)
Student:	**Řezáč Michal, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Mathematical Methods
Title:	**Heuristics in String Solving**
Category:	Software analysis and testing
Academic year:	2023/24

Assignment:

String constraint solving is useful in verification of string manipulating programs such as web-applications and in analysis of their security vulnerabilities such as cross-site scripting, SQL-injection. Modern string solvers achieve their efficiency by extending their core solving algorithms by multitude of heuristics which have either no or minimal documentation. The task is to identify and provide overview of these heuristics, analyse their impact on the efficiency of string solving, and determine those that could be used in the string solver developed in the group VefiFIT. This information will be critical for the development of this solver.

1. Study methods of string solving, especially those used in string solvers cvc5 and Z3.
2. Identify the heuristics used in string solvers (at least cvc5, Z3) and provide their overview.
3. Evaluate their impact on standard sets of benchmarks (benchmarks with basic string constraints, with length constraints, and also benchmarks with more complex string constraints such as indexof). The result should be statistical comparison of identified heuristics (and their combinations), comparing the number of solved instances and the space/time efficiency.
4. Determine and justify which heuristics could potentially be used in the string solver developed in the group VefiFIT.

Literature:

1. Roberto Amadini. 2021. A Survey on String Constraint Solving. ACM Comput. Surv. 55, 1, Article 16 (January 2023), 38 pages. https://doi.org/10.1145/3484198
2. Berzish, M. *et al.* (2021). An SMT Solver for Regular Expressions and Linear Arithmetic over String Length. In: Silva, A., Leino, K.R.M. (eds) Computer Aided Verification. CAV 2021. Lecture Notes in Computer Science(), vol 12760. Springer, Cham. https://doi.org/10.1007/978-3-030-81688-9_14
3. M. Berzish, V. Ganesh and Y. Zheng, "Z3str3: A String Solver with Theory-aware Heuristics," *2017 Formal Methods in Computer Aided Design (FMCAD)*, 2017, pp. 55-59, doi: 10.23919/FMCAD.2017.8102241.
4. Andrew Reynolds, Maverick Woo, Clark Barrett, David Brumley, Tianyi Liang, Cesare Tinelli. Scaling Up DPLL(T) String Solvers Using Context-Dependent Simplification. CAV 2017.

Requirements for the semestral defence:
1, a part of 2 and 3.

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

Supervisor:	**Síč Juraj, Mgr.**
Consultant:	Holík Lukáš, doc. Mgr., Ph.D.
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	17.5.2024
Approval date:	6.11.2023

Abstract

This work aims on identifying heuristics and strategies used in modern string solvers and evaluating their impact on the effectiveness of the solving. In particular, two solvers – cvc5 and Z3 – are examined. The thesis describes the techniques used by SMT solvers and the strategies implemented by string solvers. The evaluation of the effectiveness of the heuristics was performed by disabling them directly in the code of the tools mentioned and then evaluating the impact on solving the sets of standard benchmarks. The result of this work is summary of a set of specific heuristics and a description of the structure of the tools cvc5 and Z3. The measurements failed to demonstrate the actual impact of the heuristics identified and described.

Abstrakt

Tato práce se zaměřuje na identifikaci heuristik a strategií použitých v moderních string solverech a na vyhodnocení jejich dopadu na efektivitu řešení. Zkoumány jsou především dva solvery -– cvc5 a Z3. Práce popisuje techniky používané SMT solverech a strategie, které implementují string solvery. Vyhodnocení efektivity heuristik bylo prováděno jejich vypínáním přímo v kódu uvedených nástrojů a následným vyhodnocením dopadu na řešení standardních sad benchmarků. Výsledkem této práce je soupis sady konkrétních heuristik a popis struktury nástrojů cvc5 a Z3. Měřením se nepodařilo prokázat, jak velký skutečný dopad identifikované a popsané heuristiky mají.

Keywords

string constraints, SMT, heuristics, cvc4, cvc5, z3, string solving

Klíčová slova

řetězcová omezení, SMT, heuristiky, cvc4, cvc5, z3, string solving

Reference

ŘEZÁČ, Michal. *Heuristics in String Solving.* Brno, 2024. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Mgr. Juraj Síč

Rozšířený abstrakt

V současnosti je řetězec znaků (string) používaný významným množstvím aplikací ke komunikaci. Ať už se jedná o uložení a posílání dat ve formátech XML, JSON a jiných, protokol HTTP verze 1.1 používaný pro přenos hypertextových dokumentů nebo ovládání rozhraní mezi člověkem a počítačem pomocí příkazů či zadávání vstupu do formulářů. Operace nad řetězci a jejich zpracování může být zdrojem potenciálně velkého množství chyb nebo možných útoků jako je SQL injection. Z tohoto důvodu vznikly nástroje pro dokazování nad řetězcovými omezeními nebo také string solving.

Dokazování nad řetězcovými omezeními může pomoci s odhalováním možných nebezpečných nebo nechtěných vzorů a může sloužit k formální verifikaci. Nicméně řešení těchto omezení je z pohledu časové složitosti velmi náročné a pro obecnou teorii sekvencí je rozhodnutelnost tohoto problému stále otevřená otázka. I přes tyto vlastnosti jsou dnes nástroje (solvery) schopné vyřešit velké množství problémů a jsou využitelné ve spoustě praktických aplikací. Efektivita těchto nástrojů je dána tím, že implementují velké množství strategií, které dokážou na základě znalosti problému značně rychleji vyřešit některé jeho části. Těmto strategiím se říká také heuristiky.

Tato práce se zaměřuje na identifikaci heuristik a strategií použitých v moderních string solverech a vyhodnocení jejich dopadu na efektivitu řešení. Na základě vyhodnocené efektivity pak navrhnout heuristiky, které lze využít v nástroji vyvíjeném skupinou VeriFIT. Zkoumány jsou především dva solvery – cvc5 a Z3. Motivací pro tuto práci je, že existující nástroje pro string solving často poskytují pouze stručný či žádný popis použitých heuristik a neexistují zdroje, které by tyto heuristiky shrnovali a poskytovali nezávislé ověření jejich efektivity.

První část práce se věnuje obecným principům SAT a SMT solvingu, shrnuje standardní teorie, definuje DPLL algoritmus a jeho rozšíření o teorie, DPLL(T), včetně různých přístupů, které k implementaci DPLL(T) solveru existují. Dále shrnuje obecnou strukturu nástrojů cvc5 a Z3 a popisuje SMT-LIB a standard vytvořený touto iniciativou.

V další části práce je popis teorie sekvencí, řetězců a regulárních výrazů spolu se zadefinováním základních pojmů a funkcí této teorie. Jsou popsány string solvery zkoumaných nástrojů. Pro nástroj cvc5 je uvedeno shrnutí několika heuristických přístupů, které řeší zjednodušování vstupní formule pomocí převodu na ekvivalentní zjednodušenou podobu formule, a to s například pomocí převodu složitějších funkcí na jednodušší, dokazováním nad délkou řetězce či kontextově závislým odvozováním. Pro nástroj Z3 je uvedena strategie pro dokazování nad regulárními výrazy s pomocí symbolických derivátů. Pro oba nástroje je v práci popsaná konkrétní struktura string solveru v kódu nástroje s obecným popisem tříd a přístupy, které dané třídy implementují.

Měření efektivity jednotlivých heuristik je založeno na měření času a počtu vyřešených instancí standardních sad benchmarků, které poskytuje iniciativa SMT-LIB. Měření probíhalo nad dvěma druhy heuristik. První typ jsou heuristiky použité k syntaktickému přepisování formule do ekvivalentní podoby používající jednodušší funkce teorie řetězců. Druhým typem měřených heuristiky jsou heuristiky, které používají složitější dokazování nad vlastnostmi formule a výsledná transformace je závislá na odvozených faktech.

Výsledek měření pro první typ heuristik ukázal, že jen jednotky použitých heuristik mají vliv na větší část testovaných benchmarků. Ostatní heuristiky mají buď jen velmi malý vliv na všechny sady benchmarků a nebo významněji ovlivňují pouze nižší jednotky z nich. U druhého typu heuristik se u nástroje cvc5 jejich vliv na zvolenou část sad benchmarků neprojevil vůbec nebo byl čas řešení nižší než u nezměněné verze nástroje. To může být

způsobeno chybou měření. U nástroje Z3 modifikované nástroje nedávali validní výsledky, protože některé z nesplnitelných formulí nástroj označil jako splnitelné.

V práci se nepodařilo ukázat, které z heuristik poskytují významné zrychlení výpočtu a z toho důvodu nebylo možné navrhnout, které z heuristik by mohli být použité v nástroji vyvíjeném skupinou VeriFIT. Výsledkem práce je proto pouze shrnutí heuristik a popis struktury solverů cvc5 a Z3.

Heuristics in String Solving

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Mr. Síč. The supplementary information was provided by Mr. Holík. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....

Michal Řezáč

May 16, 2024

Contents

List of Figures

Chapter 1

Introduction

Most modern software these days usually uses strings as a form of communication – XML, JSON, and other formats are used to store and send data, the HTTP protocol is string-based, and every app with forms takes the strings from the user or commands that control the application. Especially applications that take input from user may be vulnerable to some type of attack like SQL injection, where an attacker tries to manipulate the app database, or it may generate unwanted result by inappropriate string manipulation.

String constraint solving serves not only as a tool for detecting these vulnerabilities but also as a tool for formal verification. Unfortunately, reasoning about strings constraints containing word equations, length constraint, and extended string functions is hard to solve, and the decidability of combination of all these constraints is still an open question.

Although string solving is a challenging problem, modern solvers provide good results in practical applications, due to usage of set of strategies, that simplifies the constraints to a potentially decidable set. These strategies are called *heuristics*, and implementing them helps to achieve much better efficiency. Nevertheless, heuristics does not provide a general solution of the problem, and are able to simplify or solve just some subset or instance of the given problem. Thus, the overall complexity of the problem remains unchanged. Additionally, these heuristics are generally not well documented, if at all.

Currently, active research is done in this field, where an effort is made to discover new strategies that can be used. This effort is reflected in many solvers, where the most active ones are probably the CVC5, which is made in cooperation of the University of Iowa and Stanford University, and the Z3 solver, which is a product of Microsoft Research. These solvers are the SMT solvers, which provides reasoning in many theories, not only theory of strings.

The goal of this work is to identify currently used heuristics in these solvers, provide a summary of them, and evaluate their efficiency. Based on the evaluation, justify which of them has the highest impact on the successful solving and suggest which of them can be used in the solver developed under VeriFIT.

In Chapter 2 the underlying theory of SAT and SMT solving is described. It provides a summary of the SMT-LIB standard, theories of interest in SMT, and the SAT and DPLL(T) procedures along with a possible structure of the SMT solver. At the end of the chapter, the structure of CVC5 and Z3 is provided.

A detailed description of the theory of strings is provided in Chapter 3, where is also a description of a few heuristics used in the CVC5 solver and one of the heuristics used in the Z3 solver.

Finally, in Chapter 4 the measurement is described along with the sets of benchmarks used, the benchmarking tool, and the evaluation of the results.

Chapter 2

SMT Solving

This chapter introduces the basics of solving propositional formulas known as *satisfiability solving* (or SAT solving) and its extension known as *Satisfiability Modulo Theories*, which combines SAT solving with a Theory solver. Modern SMT solving is based on the DPLL(T) algorithm, which is also introduced. And finally, the cvc5 solver is introduced.

The definitions in this chapter are taken from [16].

2.1 DPLL procedure

The Davis-Putnam-Logemann-Loveland (DPLL) procedure is an algorithm used to solve quantifier-free propositional formulas. This section provides a basic description of the procedure and its components.

2.1.1 Basic terms

Let P be a finite fixed set of propositional symbols. A $p \in P$ is called *atom*, p and $\neg p$ are *literals*. Literals can take value *true* or *false*. The propositional formula F is a string of literals connected with the following operators: unary operator *negation* $\neg$, binary operator *conjunction* $\wedge$ and binary operator *disjunction* $\vee$. A *clause* C is a disjunction of literals $l_1 \vee l_2 \vee \ldots \vee l_n$ for some $n \geq 2$. A formula in *disjunctive normal form* (DNF) is a disjunction of one or more conjunctions, in *conjunctive normal form* (CNF) is a conjunction of one or more clauses. A CNF formula can also be written as a comma-separated list of clauses $C_1, C_2 \ldots$.

An *assignment* M is set of literals such that $\{p, \neg p\} \nsubseteq M$ (any p cannot be a member of M in positive and negative form simultaneously). Literal l is *true* in M when $l \in M$, *false* in M when $\neg l \in M$, and *undefined* in M otherwise. A literal l is *defined* in M if it is either *true* or *false*. We call any assignment M *partial truth assignment* if there is some literal l from formula F such that l is undefined in M. If M is total over P when there is no literal of P is undefined in M. When assignment is $M = \{l_1, \ldots, l_n\}$, then the *negation of assignment* is an $\neg M = \{\neg l_1, \ldots, \neg l_n\}$. Clause C is true in M if there exists one of its literals in M. If all literals of C are false in M, then C is false in M. C is undefined in M otherwise. A CNF formula F is true or *satisfied* in M if all its clauses are true in M. In that case, M is a model of F, written $M \vDash F$, and is called M *model* of F. F is false in M if any of its clauses is false in M and is undefined otherwise. A formula F' is the *logical consequence* of F when all models of F are also true for F', written $F \vDash F'$, F is *entailed by* F'. Two formulas are *logically equivalent* when $F \vDash F'$ and $F' \vDash F$.

In the following text, the lowercase letter l denotes literal, the uppercase letters F, G denotes formulas, the uppercase letters C, D denote clauses, and uppercase letters M, N denote assignments. The literal in negative form $\neg l$ can also be denoted as $\bar{l}$

2.1.2 States and transitions

A *state* of the DPLL procedure is either *FailState* or pair $M \parallel F$, where F is the CNF formula and M is a (partial) assignment. The assignment M in DPPL is a sequence of literals, where each literal cannot be contained in positive and negative form simultaneously, and each literal has an *annotation* which indicates if it is a *decision* or not. M can also be considered as a set of literals, where annotation and order are ignored.

When the literal is marked as a decision literal, it is written as l^d. The empty assignment or sequence of literals is denoted as $\emptyset$. A clause C is conflicting in a state $M \parallel F, C$ if $M \vDash \neg C$.

Each DPLL procedure is modeled by a set of states along with a binary relation $\Rightarrow$ over these states, called *transition relation*. Let S, S' be states of the DPLL procedure, then $S \Rightarrow S'$ is called a *transition* from S to S'. The reflexive-transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$. Any sequence of transitions $S_1 \Rightarrow S_2, S_2 \Rightarrow S_3 \ldots$ is called *derivation* and is denoted as $S_1 \Rightarrow S_2 \Rightarrow S_3 \ldots$. Any subsequence of derivation steps is called the *subderivation*.

A *transition system* is a set of transition rules defined over some given set of states. For a given transition system R, the transition relation is denoted as $\Rightarrow_R$. If there is no transition from the state S by $\Rightarrow_R$, then S is called final with respect to R.

2.1.3 The Classical DPLL procedure

The Classical DPLL procedure consists of five transition rules. These rules are just basic rules for this system and provide a basic understanding of the procedure.

Definition 2.1 *The Classical DPLL system is the transition system Cl consisting of the following five transition rules. In this system, all literals added to M by all rules except* `Decide` *are marked as non-decision literals.*

`UnitPropagate`

$$M \parallel F, C \vee l \quad \Rightarrow Ml \parallel F, C \vee l \quad \textbf{if} \begin{cases} M \vDash \neg C \\ l \text{ is undefined in } M \end{cases}$$

`PureLiteral`

$$M \parallel F \quad \Rightarrow Ml \parallel F \quad \textbf{if} \begin{cases} l \text{ occurs in some clause of } F \\ \neg l \text{ occurs in no clause of } F \\ l \text{ is undefined in } M \end{cases}$$

`Decide`

$$M \parallel F \quad \Rightarrow M l^d \parallel F \quad \textbf{if} \begin{cases} l \text{ or } \neg l \text{ occurs in } F \\ l \text{ is undefined in } M \end{cases}$$

`Fail`

$$M \parallel F, C \quad \Rightarrow \mathit{FailState} \quad \textbf{if} \begin{cases} M \vDash \neg C \\ \text{there is no decision literal in } M \end{cases}$$

`Backtrack`

$$M l^d N \parallel F, C \quad \Rightarrow M \neg l \parallel F, C \quad \textbf{if} \begin{cases} M l^d N \vDash \neg C \\ \text{there is no decision literal in } N \end{cases}$$

The input formula F can be decided in the transition system Cl by applying these rules and generating a derivation $S_0 \Rightarrow S_1 \Rightarrow \ldots \Rightarrow S_n$, where S_n is the final state with respect to Cl. For S_n is a formula unsatisfiable if, and only if, S_n is *FailState*, or S_n is of the form $M \parallel F$ and then M is the model of F. The second part of a state remains unchanged in the classical DPLL procedure.

Here is a brief explanation of the rules. The input formula F is considered in CNF form. To satisfy such a formula, all its clauses have to be true.

- `UnitPropagate` – if there is literal l that is undefined in M and all other literals of a clause are already false, then l has to be true.
- `PureLiteral` – when a literal l appears in a formula only in positive or negative form, then l is called *pure*, and l ($\neg l$ respectively) is added to M.
- `Decide` – if an l appears in the formula in both positive and negative form, then the literal is added to M and marked as a decision literal. This denotes that, if Ml cannot be extended to be a model of F then the alternative extension $M\neg l$ has to be still considered.
- `Fail` – if there is some conflicting clause and M does not contain any decision literal, then *FailState* is produced.
- `Backtrack` – if there is a conflicting clause detected and `Fail` does not apply, then this rule backtracks to the last decision literal Ml^dN and replaces it with its negation and removes any subsequent literals N, where N does not contain any decision literal. Then the assignment has a form of $M\neg l$, where $\neg l$ is no longer marked as decision literal.

These rules are applied in exactly the same order as was introduced above (`UnitPropagate` have the highest priority). When the DPLL procedure cannot apply the rule, then it tries to apply its successor. When all clauses are true, then a model of a formula is returned, or some clause is false, and there is no decision literal. In that case, *FailState* is returned. Based on that observation, it can be shown, that the DPLL procedure stops for all formulas.

2.1.4 Modern DPLL procedure

Today, modern SAT solvers do not implement the classical DPLL procedure. To gain higher efficiency, there are some modifications to the algorithm.

Usually in modern solvers is the `PureLiteral` rule applied as part of formula preprocessing. Therefore, this rule is no longer part of the DPLL procedure itself. Another rule, which is modified, is `Backtrack`. Typically, the backtrack rule chooses the last decision literal to replace. But in some cases it is more effective, to select another decision literal. This approach is called *backjumping*. The modified algorithm with backjumping is called *Basic DPLL System*

Definition 2.2 *The Basic DPLL system is, is a four-rule transition system B consisting of the rules* `UnitPropagate`, `Decide`, `Fail` *from Classical DPLL, and the following* `Backjump` *rule:*

`Backjump`

$$Ml^dN \parallel F, C \quad \Rightarrow Ml' \parallel F, C \quad \textbf{if} \begin{cases} Ml^dN \vDash \neg C \text{ and there is some clause} \\ C' \vee l' \text{ such that:} \\ \quad F, C \vDash C' \vee l' \text{ and } M \vDash \neg C', \\ \quad l' \text{ is undefined in } M, \text{ and} \\ \quad l' \text{ or } \neg l' \text{ occurs in } F \text{ or in } Ml^dN \end{cases}$$

The C clause in `Backjump` is called *conflicting* clause, and clause $C' \vee l'$ is the *backjump* clause. The choice of backjump clause is illustrated by the following example [16].

Example 2.1. This sequence of states is generated by the Classical DPLL procedure without the `PureLiteral` rule.

$$\begin{aligned}
\emptyset \parallel (\overline{1} \vee 2), (\overline{3} \vee 4), (\overline{5} \vee \overline{6}), (1 \vee 3 \vee 5), (\overline{2} \vee \overline{5} \vee 6) &\Rightarrow_{Cl} && \texttt{(Decide)} \\
1^d \parallel (\overline{1} \vee 2), (\overline{3} \vee 4), (\overline{5} \vee \overline{6}), (1 \vee 3 \vee 5), (\overline{2} \vee \overline{5} \vee 6) &\Rightarrow_{Cl} && \texttt{(UnitPropagate)} \\
1^d\ 2 \parallel (\overline{1} \vee 2), (\overline{3} \vee 4), (\overline{5} \vee \overline{6}), (1 \vee 3 \vee 5), (\overline{2} \vee \overline{5} \vee 6) &\Rightarrow_{Cl} && \texttt{(Decide)} \\
1^d\ 2\ 3^d \parallel (\overline{1} \vee 2), (\overline{3} \vee 4), (\overline{5} \vee \overline{6}), (1 \vee 3 \vee 5), (\overline{2} \vee \overline{5} \vee 6) &\Rightarrow_{Cl} && \texttt{(UnitPropagate)} \\
1^d\ 2\ 3^d\ 4 \parallel (\overline{1} \vee 2), (\overline{3} \vee 4), (\overline{5} \vee \overline{6}), (1 \vee 3 \vee 5), (\overline{2} \vee \overline{5} \vee 6) &\Rightarrow_{Cl} && \texttt{(Decide)} \\
1^d\ 2\ 3^d\ 4\ 5^d \parallel (\overline{1} \vee 2), (\overline{3} \vee 4), (\overline{5} \vee \overline{6}), (1 \vee 3 \vee 5), (\overline{2} \vee \overline{5} \vee 6) &\Rightarrow_{Cl} && \texttt{(UnitPropagate)} \\
1^d\ 2\ 3^d\ 4\ 5^d\ \overline{6} \parallel (\overline{1} \vee 2), (\overline{3} \vee 4), (\overline{5} \vee \overline{6}), (1 \vee 3 \vee 5), (\overline{2} \vee \overline{5} \vee 6) &\Rightarrow_{Cl} && \texttt{(Backtrack)} \\
1^d\ 2\ 3^d\ 4\ \overline{5} \parallel (\overline{1} \vee 2), (\overline{3} \vee 4), (\overline{5} \vee \overline{6}), (1 \vee 3 \vee 5), (\overline{2} \vee \overline{5} \vee 6) &
\end{aligned}$$

The clause $\overline{2} \vee \overline{5} \vee 6$ is false in model $1^d\ 2\ 3^d\ 4\ 5^d\ \overline{6}$ before the `Backtrack` step, thus it is a conflicting clause. It is a consequence of the unit propagation 2 of the decision 1^d, together with the decision 5^d and its unit propagation $\overline{6}$. Here, instead of `Backtrack`, a `Backjump` can be used.

As can be observed, the decision 1^d is incompatible with the decision 5^d, therefore the set of clauses entails $\overline{1} \vee \overline{5}$. This clause can be obtained by an analysis starting with the conflicting clause. $\overline{2} \vee \overline{5} \vee 6$. This clause is in contradiction with the $\overline{5} \vee \overline{6}$ clause, leading to the inference of clause $\overline{2} \vee \overline{5}$, which can be used as a backjump clause. Moreover, additional

analysis shows that the latter clause is in contradiction with $\overline{1} \vee 2$. From these two coflicting clauses, the clause $\overline{1} \vee \overline{5}$ can be inferred, which can also be used to guide the backjump.

With that entailed backjump clauses `Backjump` goes back to its decision level and adds unit propagated literal. In this case, clause $\overline{2} \vee \overline{5}$ could be used along with `Backjump` replacing `Backtrack`, to get partial assignment in form $1^d\ 2\ \overline{5}$.

Moreover, `Backtrack` rule can be considered a special case of `Backjump` rule.

`Backjump` rule based on conflicting clause is called *conflict driven*. Compared to `Backtrack` rule, it can backtrack further in the assignment by analyzing the reason that produced the conflicting clause. Due to that, it undoes several decisions, skipping decisions which are irrelevant to the conflict. In the previous example, the conflicting clause is $6 \vee \overline{5} \vee \overline{2}$, so it skips literals $3^d 4$, which does not affect the evaluation of this clause.

As an extension of the `Backjump` rule, the backjump clause can be added to the evaluated formula. This approach is called *conflict-driven learning* and is implemented by two new rules: `Learn` and `Forget`.

Definition 2.3 *The DPLL system with learning, denoted by L, consists of the four transition rules of the Basic DPLL system and the two additional rules:*

`Learn`

$$M \parallel F \quad \Rightarrow M \parallel F, C \quad \textbf{if} \begin{cases} \text{each atom of } C \text{ occurs in } F \text{ or in } M \\ F \models C \end{cases}$$

`Forget`

$$M \parallel F, C \quad \Rightarrow M \parallel F \quad \textbf{if} \begin{cases} F \models C \end{cases}$$

In any application step of `Learn`, clause C is said to be *learned* if it did not already belong to F. Similarly, it is said to be *forgotten* by `Forget`.

The `Learn` rule allows *learning* of the clause C which is entailed by F as long as all atoms of C occur in F or M. Similarly, the `Forget` rule allows removal of any clause from F, whenever F entails C. This means that not only the learned clauses can be forgotten.

As can be observed in Example 1, the learning clause $\overline{2} \vee \overline{5}$ will allow application of the `UnitPropagate` rule, whenever 2 or 5 appears in the assignment. In addition, learning such clauses will prevent the repetition of computing the same or similar backjump clauses, which leads to improving the performance of the solver.

Because learning clauses are aimed at preventing future conflicts when the conflicts are not likely to be found, the clause can be removed. This can be tracked by the relevance or activity of the clause. Activity can be measured by the number of times it is used as a unit or conflicting clause and removed when the number drops below a given threshold.

The following theorems show the correctness of DPLL systems with learning. All proofs can be found in [16].

First, consider $\emptyset \parallel F \Rightarrow_L \ldots \Rightarrow_L S_n$, where S_n is the final state. The final state does not always exist, due to the possible infinite application of the rules `Learn` and `Forget`. Thus, the infinite application of these rules is avoided.

Lemma 2.1 *If $\emptyset \parallel F \Rightarrow_L^* M \parallel G$, then all the following hold.*

(1) All the atoms in M and all the atoms in G are atom of F

(2) M contains no literal more than once and is indeed an assignment, i.e., it contains no pair of literals of the form p and $\neg p$.

(3) G is logically equivalent to F.

(4) If M is of the form $M_0 l_1 M_1 \dots l_n M_n$, where $l1, \dots, ln$ are all the decision literals of M, then $F, l_1, \dots, l_i \vDash M_i$ for all i in $0 \dots n$

Lemma 2.2 *Assume that $\emptyset \parallel F \Rightarrow_L^* M \parallel F'$ and that $M \vDash \neg C$ for some clause C in F'. Then either* `Fail` *or* `Backjump` *applies to $M \parallel F'$.*

Lemma 2.3 *If $\emptyset \parallel F \Rightarrow_L^* S$, and S is final with respect to Basic DPLL, then S is either FailState, or it is in the form $M \parallel F'$, where*

(1) all literals of F are defined in M,

(2) there is no clause C in F' such that $M \vDash \neg C$, and

(3) M is a model of F

Theorem 2.1 *There are no infinite derivations of the form $\emptyset \parallel F \Rightarrow_B S_1 \Rightarrow_B \dots$*

Theorem 2.2 *Every derivation $\emptyset \parallel F \Rightarrow_L S_1 \Rightarrow_L \dots$ by the DPLL system with learning is finite if it contains no infinite subderivations consisting of only* `Learn` *and* `Forget` *steps.*

Theorem 2.3 *If $\emptyset \parallel F \Rightarrow_L^* S$ where S is final with respect to Basic DPLL, then*

(1) S is FailState if, and only if, F is unsatisfiable.

(2) If S has the form $M \parallel F'$, then M is the model of F

The last theorem also holds for the relation $\Rightarrow_B$.

The last rule used by Modern DPLL is the `Restart` rule. In some cases, the solver does not make any significant progress in solving the given formula. Then `Restart` is applied. The main idea of restarting is that the solver has additional information on the solved formula, which was obtained by applying the `Learn` rule. In the next run, more suitable heuristics for `Decision` can be used. The *Decision heuristics* try to find which form of literal, positive, or negative, is more suitable to successfully solve the given formula. The combination of rules `Restart` and `Learn` was shown to increase the efficiency of the algorithm in both theoretical and practical ways.

The `Restart` rule is defined as follows:

Definition 2.4 *The* `Restart` *rule is:*

$$M \parallel F \Rightarrow \emptyset \parallel F$$

As can be seen, it simply discards any partial assignment obtained before.

As mentioned above, `Restart` is applied whenever the solver does not make any significant progress. This metric can be modeled simply by setting an expected number of steps to solve the given formula. To ensure that the DPLL procedure with restart always ends and will not apply the `Restart` infinitely, the expected number of steps is increased whenever `Restart` is applied. This can be formalized as follows:

Definition 2.5 *Consider a derivation of the DPLL system with learning extended with the* `Restart` *rule. We say that* `Restart` *has increasing periodicity in the derivation if, for each subderivation* $S_i \Rightarrow \ldots \Rightarrow S_j \Rightarrow \ldots \Rightarrow S_k$*, where the steps producing* S_i, S_j, S_k *are the only* `Restart` *steps, the number of Basic DPLL steps in* $S_i \Rightarrow \ldots \Rightarrow S_j$ *is strictly smaller than in* $S_j \Rightarrow \ldots \Rightarrow S_k$*.*

Theorem 2.4 *Any derivation* $\emptyset \parallel F \Rightarrow S_1 \Rightarrow \ldots$ *by the transition system* L *extended with the* `Restart` *rule is finite if it contains no infinite subderivation consisting of only* `Learn`*,* `Restart` *steps, and* `Restart` *has increasing periodicity in it.*

The proof of Theorem 2.4 can be found in [16].

In this section, definitions of all DPLLs are provided. For clarity, the DPLL system with learning is also called CDCL – *Conflict-Driven Clause Learning* (sometimes also Constraint-Driven) [15].

2.2 Basic principles of SMT solving

In practical problem solving, simply taking a problem and converting it into a propositional formula is not sufficient to obtain a compliant result. Coding problems into propositional formulas can be a fairly difficult process if possible. For example, consider the classical Sudoku problem. To encode it into a propositional formula, it is necessary to create nine variables for each cell, to express each of the possible values, and to create clauses that specify given constraints. To create an extended version of Sudoku, the proper number of variables and clauses is needed to express all constraints properly. This approach is even more complicated for more complex problems and leads to a variable explosion.

Instead of converting a problem into a propositional formula, it is more suitable to specify constraints on a given problem. In that case, variables do not express only Boolean values, but can be considered as a set of values in a given theory. For the Sudoku problem, each variable stands for a number from 1 to 9, and constraints for each column, row, and square are specified just above them, leading to a more compact description of the given problem. To solve such problems, the SMT solver is used.

The *Satisfiability Modulo Theories* solver is basically an SAT solver extended by a theory checker. Its input formula can be specified using first-order logic and theory-specific elements. This family of solvers works basically as follows: The input formula is evaluated by the SAT solver simply by considering each theory expression as literal, and then the theory checker checks if a given assignment is also valid in its theory.

Some SMT solvers today use the architecture called DPLL(T). This notation says that a DPLL-based solver is combined with a given theory T. That also provides an interface for the SAT solver, which can be combined with any available theory T. In this section an overview of most used theories, to demonstrate a decision complexity over them. A comprehensive description of the principles of SMT solvers is provided. This work aims at examining the string heuristics used in two solvers, CVC5 and Z3. At the end of the section, these two solvers are briefly introduced followed by description of SMT-LIB standard.

The Sections 2.2.3, 2.2.4 and 2.2.5 are based on [5, 6, 16]

2.2.1 Basic terms

The following section uses the definition from section 2.1.1 with the difference in the definition of set P. P is fixed set of *ground* first-order atoms. Such atoms are variable-free. The

difference between formulas, formula φ is the first-order formula that contains function symbols, propositional symbols, and equivalencies between ground terms. The formula F contains only literals $p, \neg p$, where $p \in P$.

In addition to propositional logic, some notion of first-order logic is also used. A *theory* T is a set of ground (variable-free) first-order formulas, where all symbols in the formula are symbols of the theory signature. These formulas A formula F is *T-consistent* or *T-satisfiable* if $F \wedge T$ is satisfiable in the first-order sense. When it is not satisfiable, then F is *T-unsatisfiable* or *T-inconsistent*.

The partial assignment M can be considered as a conjunction of literals and thus as a formula. If F is the formula, M is the T-consistent partial assignment and $M \vDash F$, then M is called the *T-model* of F. If F and G are formulas, then F *entails* G *in* T, written $F \vDash_T G$, when $F \wedge \neg G$ is T-inconsistent. If $F \vDash_T G$ and $G \vDash_T F$, then F and G are *T-equivalent*. A *theory lemma* is clause C, where $\emptyset \vDash_T C$.

The SMT problem for a given formula F and the theory T is to determine if F is T-satisfiable. Equivalently, it can be asked if F has a T-model.

The *ground* CNF formula F is the variable-free and quantifier-free formula. The SMT problem will be considered only for the ground formulas. Such formulas may contain *free* constants (constant symbols that are not in the signature of a theory T). Such constants can be seen as existential variables for the purpose of satisfiability. The function and the predicate symbols in the formulas, in addition to the free constants, all come from the signature of a theory T. In the following sections, a formula is meant as a formula with all previous restrictions considered.

Also, by theory T is meant only theories that are decidable in the case of T-satisfiability of conjunctions of the ground literals. The decision procedure for this problem is called *T-solver*.

2.2.2 Theory overview

The SMT solvers can be used to solve problems in a variety of theories. Certain solvers are specialized only for a few of them, while others provide a robust set of theories. Below is a brief overview of the most commonly used theories.

Equality and Uninterpreted Functions (EUF) is the most general case of the theory. This theory has no axioms due to what is also called an empty theory. The only possible operator is an equality operator $=$. The uninterpreted functions are just abstraction without any functionality. This helps in modeling a system without an unnecessary level of detail. An example of a formula in a given theory may be $f(a) = b, g(b) = a, f(g(b)) = b$, where a, b are constant symbols, and f, g are uninterpreted functions. This theory is decidable in polynomial time using the algorithm called *Congruence closure* [5].

Real and Integer Arithmetic theories, uses the signature $\{+, -, *, \leq\}$, where $\{+, -, *\}$ are function symbols, and $\leq$ is a predicate symbol. For integers, arithmetic with this signature is in general undecidable [6]. For practical purposes, often a decidable fragment of integer arithmetic is used. It is called *Pressburger arithmetic*, where the use of multiplication $*$ is excluded or restricted to the case where one of the multiplication operands must be a constant number. Against it, for real numbers, this theory is decidable in polynomial time without the need to limit the signature [5]. Using SMT solvers in practical scenarios leads to finding many solutions in integers and enhances the effectiveness for reals.

One fragment of arithmetic, called a *Difference Logic* theory, uses the atomic formulas in the form $a - b \bowtie c$, where a, b are (uninterpreted) constants or variables, c is a constant, and $\bowtie \in \{=, \leq\}$. Based on an instance of difference logic, a c can be both an integral or a real constant, giving an integer difference logic or a real difference logic, respectively. This fragment can be solved in polynomial time [6].

Bit Vectors theory using reasoning on fixed-size bit vectors. This can be used to verify hardware components or for a compact transcript of some of the propositional formulas. The signature may contain operations *concatenation*, *extraction* of one bit from the vector, bitwise logic operation, and arithmetic operations. By a reduction to a SAT problem, it may be shown that this problem is NP-complete [5]. But in the case where only concatenation and extraction over equations are used, a formula can be solved in polynomial time. However, by adding more extensions (e.g., disequalities), this problem can explode up to NEXPTIME-complete complexity [6].

Arrays is theory of arrays, indices, and elements that have signature $\{\texttt{read}, \texttt{write}\}$, where the result of `read` is the element and the result of `write` is an array. The basic axioms of this theory are:

$$\forall a \forall i \forall e(\texttt{read}(\texttt{write}(a, i, e), i) = e)$$
$$\forall a \forall i, j \forall e(i \neq j \rightarrow \texttt{read}(\texttt{write}(a, i, e), j) = \texttt{read}(a, j))$$

In SMT-LIB standard is a theory of arrays with extensionality, which needs one more axiom defined as following:

$$\forall a \forall b(\forall i(\texttt{read}(a, i) = \texttt{read}(b, i))) \rightarrow a = b$$

This theory can be used to model an array in program or for modeling a memory, which may significantly decrease complexity due to that the size of memory is defined by number of accesses and not by actual size of modeled memory. In general, this theory is undecidable [6].

Strings and Regular expression theory serves for reasoning about words and languages. It is usually combined with a linear arithmetic fragment, which enables reasoning about word length constraints. The theory of strings with comprehensive signature is undecidable, but some restricted fragments are decidable [13]. More parts of this theory are described in Chapter 3.

The SMT-LIB standard also contains a few other theories. Theory of *floating point numbers* based on IEEE standard 754-2008, theory of *Combined Integer and Real arithmetic* and so called *Core* theory, which defines basic boolean operators.

In the following sections, two approaches on how to implement an SMT solver based on the combination of a theory solver and an SAT solver and internal communication between them, is introduced – *lazy* and *eager* techniques. For both of these techniques, an SAT solver creates the core of the solver; however, some degree of variability is possible, due to easy combination of the SAT and a Theory solver. There also exist a bit different technique of SMT solving typically used for (non)linear arithmetic called MCSAT. A further description of the latter can be found in the literature [15].

2.2.3 Eager SMT techniques

The main idea of eager techniques is to take an input formula and translate it into a propositional CNF formula by a satisfiability-preserving transformation, which satisfiability can then be checked by the SAT solver. Due to that, the SMT solvers using this technique are often referred to as *SAT based*.

The main advantage of this technique is that the best available SAT solver can be used directly. Therefore, the efficiency of this approach increases whenever a new faster SAT solver appears. But this also leads to the biggest disadvantage of eager techniques, which is that there is a need to have an efficient and complex translation for each theory. In addition, for each theory, there can be a unique approach to translating it into CNF formula.

The eager technique correctness depends on the correctness of both a translator of the theory solver and the SAT solver. Also, there is a problem with running out of memory or time, which is based on the full translation of the input formula into the complete CNF formula.

Some of the eager techniques are listed below. Generally speaking, each technique is unique for theory and is challenging to classify them into broader categories.

For theories using lambda functions, *elimination of the lambda expression* can be used which is based on *beta reductions*. Such a reduction is a simple subsitution of argument variables with given term, which can cause the exponential blow-up in size of input formula. However, in practical cases, the increase in formula size is typically only linear [5].

The theory of equality with uninterpreted functions uses *elimination of function application*. One of the methods used is the *Ackermann's method*, where the function symbols $f_i(x_i)$ for $i = 1 \dots n$ are substituted in the input formula φ with new constant symbols f_i, producing formula F. Then the formula $F \wedge \bigwedge_{i=0}^{n-1} \bigwedge_{j=i}^{n} (x_i = x_j \implies f_i = f_j)$ is satisfiable iff formula φ is satisfiable in the EUF theory [6]. This method can be extended for any non-nulary function application and also applies for non-nulary predicate symbols.

Few methods can be grouped by the principle of *bounding of problem size*. For simplicity, these methods bound possible values to some subset of values, where a solution can be found. For example, in Integer Arithmetic Theory, the method of *Small Domain Encoding* [5] can be used. This method bounds the state space of the formula based on the maximal coefficient and constant, the number of constraints, and the number of all constraints variables. Another example can be shown in the Character String Theory, where the size of each string variable can be bounded, and also an alphabet can be reduced to some subset of symbols, which are sufficient for finding a solution [14]. In addition, this method uses the refinement of the bounds during solving.

The given techniques serve only as an example of part of the eager approach, and typically all such techniques are used as some part of the eager translation and cannot be used as standalone solvers.

In the following section, the opposing lazy approach will be described.

2.2.4 Lazy SMT techniques

An alternative to the eager technique can be used the *lazy* technique. This approach uses a greater integration of the SAT solver and the theory solver. Each atom in the input formula is seen as a propositional symbol. Then such a formula F is given to the SAT solver, which determines its theory-independent propositional satisfiability. If the formula is unsatisfiable, then it is also T-unsatisfiable. Otherwise, the SAT solver returns model M of the formula.

Then, the theory solver checks the consistency of a given model. If model M is T-consistent, then it is the model of the input formula. Otherwise, the solver generates a ground clause, which is a logical consequence of T and contradicts that inconsistent model. Then this lemma is added to F and again given to the SAT solver. This procedure is repeated until a T-consistent model or an unsatisfiable formula is obtained.

To successfully implement a lazy solver, the theory solver has to implement some features, which lead to effective cooperation of T-solver and SAT solver [5]. A *model generation*, mentioned above, leads to producing *T-model*, which witness the consistency of the input formula. Today, it is also an important property of modern solvers that the model of a given formula is returned. *Generating a theory conflict set* is used to obtain the conflict set, which can be learned by the SAT solver and/or used as a backjumping clause. Also important for the T-solver is to keep an actual state of computation. This property ensures that for newly obtained literals T-solver does not have to check the already verified parts of the obtained model and is called *Incrementality*. Keeping the actual state also allows T-solver to add undo steps, making *Bactrackability* possible. And also a *Deduction of unassigned literals* allows T-solver to deduce new literals from a given partial assignment obtained during computation.

The theory solver with described features can be relatively easily combined with any SAT solver using the lazy approach. Also, when the SAT solver is DPLL-based, many refinements exist, which increases the effectivity of the solver. Some of them are described below.

Incremental T-solver is based on checking the T-consistency during the execution of the DPLL procedure, not only after the formula proposition model is generated. The incremental checking causes the inconsistent literal to be found much earlier in the process, which increases the efficiency of the solver. Detecting T-inconsistencies can be done whenever a new literal is generated, or, if it is too expensive, at regular intervals. After the T-inconsistency is detected, the conflicting clause is learned, and the solver restarts. To gain greater effectivity, the implementation of an incremental T-solver must be specific. The T-solver has to process the new coming literal l generally faster than reprocess the complete previous input with the new literal l.

Online SAT-solver can be used in cooperation with an incremental T-solver. When the T-solver detects T-inconsistency, the DPLL procedure can be asked to backtrack to a point where the assignment was T-consistent, instead of restarting and building the complete assignment again. From the T-inconsistent assignment a theory lemma can be generated, which can be added to the input formula and used for `Backjump` rule. There is a formal proof that the theory lemma can be forgotten after using the `Backjump` rule, while T-consistent formula will still be found if it exists [16]. However, keeping such a lemma can increase the effectiveness of the solver, due to the preceding possible future conflicts. The most useful lemmas, in terms of the solver effectivity, were shown to be the small ones.

Theory propagation also implements a way in which the T-solver communicates with an SAT solver. Instead of only checking if a given assignment is T-consistent, this approach additionally allows us to guide the next state of the DPLL-based SAT solver. If, for a given state $M \parallel F$, the T-solver detects that for some literal l the $M \vDash_T l$ holds, then guides the SAT-solver to move to state $Ml \parallel F$.

It has been shown that the *theory propagation* can be effectively implemented in SMT solvers and makes a major improvement in SMT solving performance [16].

Another improvement can be obtained by applying *Theory propagation* exhaustively. All possible *Theory propagations* are applied before the `Decide` rule or even before calling the SAT solver, because a new state $Ml \parallel F$ may possibly entail a new literal l'. This approach can remove most of the unwanted redundancy of theory information, which can be generated in a system without *Theory propagation* by learning new theory lemmas or conflict clauses.

The ideas provided are used to create an abstract framework, which is described in the following section.

2.2.5 Abstract DPLL(T)

This section provides one application of the lazy approach. The following definition provides a system that solves the formula in a propositional way while checking the consistency with a given theory. The first-order theory entailment $\vDash_T$ is used by all rules.

Definition 2.6 *An Abstract DPLL(T) is system consisting of rules* `UnitPropagate`*,* `Decide`*,* `Fail` *and* `Restart` *of the Basic DPLL system and defines four new theory rules:*

`T-Propagate`

$$M \parallel F \quad \Rightarrow M, l \parallel F \quad \textbf{if} \begin{cases} M \vDash_T l \\ l \text{ or } \neg l \text{ occurs in } F \\ l \text{ is undefined in } M \end{cases}$$

`T-Learn`

$$M \parallel F \quad \Rightarrow M \parallel F, C \quad \textbf{if} \begin{cases} \text{each atom of } C \text{ occurs in } F \text{ or in } M \\ F \vDash_T C \end{cases}$$

`T-Forget`

$$M \parallel F, C \quad \Rightarrow M \parallel F \quad \textbf{if} \begin{cases} F \vDash_T C \end{cases}$$

`T-Backjump`

$$Ml^d N \parallel F, C \quad \Rightarrow Ml' \parallel F, C \quad \textbf{if} \begin{cases} Ml^d N \vDash_T \neg C \text{ and there is some clause} \\ C' \vee l' \text{ such that:} \\ \quad F, C \vDash_T C' \vee l' \text{ and } M \vDash \neg C', \\ \quad l' \text{ is undefined in } M, \text{ and} \\ \quad l' \text{ or } \neg l' \text{ occurs in } F \text{ or in } Ml^d N \end{cases}$$

This scheme is the most common application of the DPLL(T) schema, which were shown to be one of the most effective rules set in a given framework [5]. An Abstract DPLL(T) procedure returns either a model of the given formula or ends in *FailState*, which marks input formula as Theory inconsistent (or Theory unsatisfiable).

- *T*`-Propagate` is an practical application of *Theory propagation*. This rule defines a new literal l, which is yet undefined in the partial model M, when M entails l in a given theory. This leads to performance improvement caused by pruning the search space.

- T-`Learn` and T-`Forget` adds a clause to a formula F or removes it, respectively. Each clause does not provide new literals, so it does not expand a search space. Or in case of forgetting, the resulting model M of the formula remains untouched. The T-`Learn` rule also provides a pruning of the search space by adding a new clause entailed by theory. Even if the model is empty, before the procedure starts, clauses can be learned to provide some type of early pruning of search space. In addition, this learning can be used to learn backjump clauses.

- T-`Backjump` makes a backjump step based on the theory conflict. That leads to more precise backjumping. The clause C is a conflicting clause and the clause $C' \vee l'$ is a backjumping clause. The literal l' in the backjumping clause is the consequence of the decision literal l^d, which causes the conflict and which will be propagated in the model after the backjump. As can be seen, this rule uses both the theory entailment $\vDash_T$ along with the propositional satisfiability $\vDash$.

There are also more approaches, which are complete from the view of termination of a given algorithm. All of them differs mainly on the set of theory rules that are used.

A systems with only T-`learn` rule are sufficient. The basic implementation waits for the total assignment from the DPPL procedure, which is then checked for T-consistency. If T-inconsistency is detected, a conflicting clause is learned, and the system is restarted. An improvement may be made here with the incrementality of the solver, which may detect T-inconsistency much earlier in the process.

By adding T-`backjump` rule to the system with learning, an Online SAT-solver will be created. When T-inconsistency is detected, only backjump is performed, which avoids solving the untouched part of the formula again.

One of the approaches uses only T-`propagate` with the DPLL system. This system also terminates and increases effectivity, because T-`propagate` rule substitutes the `UnitPropagate` rule potentially in a more effective manner [16].

Abstract DPLL(T) framework is combination of all mentioned approaches.

The practical implementation of a DPLL(T) framework can be realized by a combination of the SAT engine DPLL(x), and theory solver (T-solver), which provides an interface for the DPLL engine. The DPLL(x) engine is theory independent and any theory can be provided as a parameter x. In this particular implementation, a DPLL(x) engine calls T-solver, to validate an actual assignment, and to get information about a T-consistency.

An interface of the T-solver shall provide a mechanism to notify that a literal was set to true and a way to undo the last notifications. Method that checks if given assignment M is T-consistent. A method to identify input literals that are a T-consequence of actual assignment and are not defined yet in the assignment and a method which returns an explanation of a theory-propagated literal. A detailed description of these methods and their implementation can be found in the literature [16].

2.2.6 SMT solvers

There are many tools available[1] that provide SMT solving of some kind. This work aims at two of them, CVC5 and Z3. These solvers have a similar architecture, both provide a number of theories, and also both support an STM-lib format. This section provides a description of these solvers. SMT-lib format is described in Section 2.2.7.

[1] More solvers, either under active development or not maintained futher, can be found on https://smt-lib.org/solvers.shtml

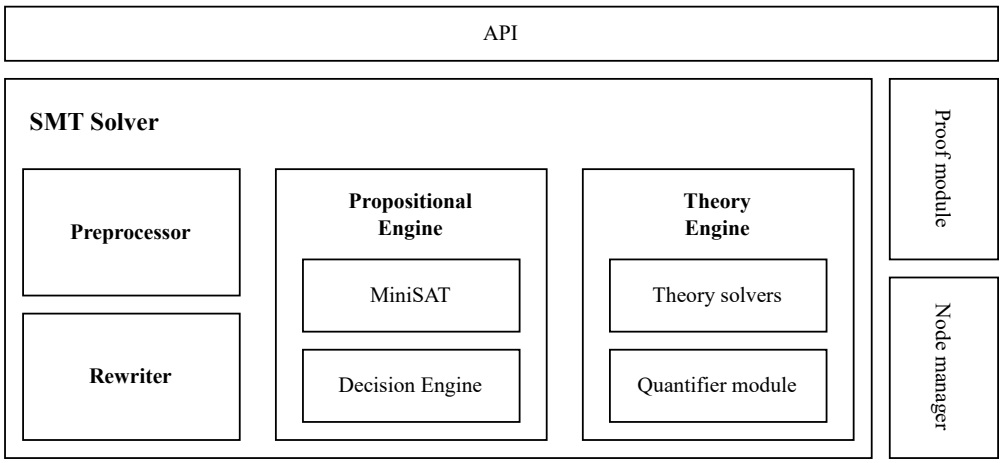

Figure 2.1: A simplified schema of CVC5 solver taken from [2].

CVC5 [2] is a solver from the family of *cooperating validity checkers*, supports reasoning over quantifier-free and quantified formulas, which is built on a CVC4 codebase. It is based on a CDCL(T) framework, which using a MiniSAT solver, customized for the purpose of CVC5. This solver supports input in SMT-libv2, SyGuS2 and TPTP textual format, and a C++ API with bindings for Python and Java. The main components of the solver are *Preprocessor*, *Rewriter*, *Theory Engine* and *Propositional Engine*.

Preprocessor provides set of rewrites for the input formula to obtain *normalized* and simplified formula for the solver. Some preprocessing steps are mandatory for the solver; some of them are optional and may be disabled. All transformations are satisfiability-preserving.

Propositional Engine is internally divided into subcomponents, which provides the management and solving of Boolean abstraction of the input formula. The input Boolean abstraction is internally transformed into CNF form. The main component of this engine is the MiniSAT solver, which serves as core of DPLL(T) framework. The engine notifies the Theory solver about each new assigned literal, which may perform T-consistency checking on actual partial assignment. In complete assignment, the T-consistency is verified immediately, producing SAT and a model or theory conflict clause.

Rewriter module provides the transformation of actual terms during solving into equivalent normalized terms with a given set of rules. As in the Preprocessor, some of the rules are required, while some can be disabled by the user. A cache is kept in the module to prevent repeated processing of terms.

Theory engine is responsible for proper checking T-consistency via distributing literals retrieved from the Propositional module to the appropriate theory solver. The engine is also responsible for communication with the Quantifier module when an abstraction over quantifiers was changed. When a combination of theories is used, the engine also provides the Combination Engine, which is responsible for coordination of the required theory solvers.

Theory solvers are a set of modules, each responsible for a given theory. Each solver needs an Equality engine which is able to detect equality conflict with congruence closure algorithm. Also, each information produced by solver (e.g. propagated literals, conflict clauses) is provided to system through Theory inference manager, which is responsible for caching, proof production, rewriting lemmas and statistic collection. There are provided theory solvers for (Non-)Linear Arithmetic and Floating-Point Arithmetic theories, theory

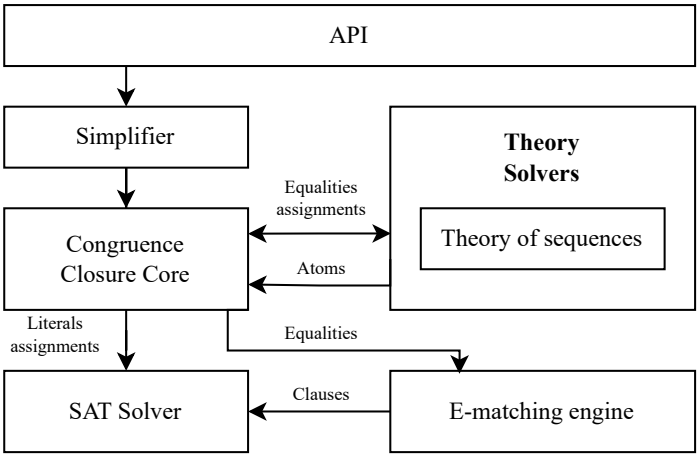

Figure 2.2: A simplified schema of Z3 solver taken from [11].

of Arrays, Bit-vectors, Datatypes, Uniterpreted Functions, Sets and Relations, Separation Logic, Quantifiers and Strings and Sequences.

String and Sequences module is implemented as multiple layered components. It is a compound of the solver of length constraints and the word equation solver, effective handling of conversion between strings and integers, extended function simplification, generating derivatives of regular expressions. The important part is the aggressive simplification of strings and the eager detection of conflicts via computing congruence closure and constant suffixes and prefixes of string terms.

For practical solving, an effective way to manage the input is necessary. The input formula is translated into the form of a directed acyclic graph, where each node represents one formula or term. The nodes are managed with the *Node manager*. Each node is saved in the structure just once. Every new node with exactly the same structure in the input formula is just referenced, which saves memory and improves performance in case of equality checks.

CVC5 solver also provides a Proof module, which is able to produce proofs in various formats, which can be verified by automated proof checkers and assistants.

Z3 [11, 7, 8] is CDCL(T) based SMT solver that aims at software verification and software analysis. The used SAT solver is DPLL-based. Also, Z3 provides other engines, which are used for particular theories. For non-linear arithmetic a NLSAT engine is used, for solving Constrained Horn Clauses a SPACER is used and for solving satisfiability of quantified formulas a QSAT algorithm is used. Default input format is textual SMT-LIBv2 format, also API is provided for C/C++ (and bindings to other languages via C API – e.g. Python, Julia, Java), .NET and OCaml. The description of the main components is provided below.

Simplifier perform an incomplete simplification of input formulas, using standard algebraic reduction rules. The simplified formula is equisatisfiable with the former formula. It may use a set of different simplifying tactics based on a solved theory.

Congruence Closure Core serves as the core of the Z3 solver. It receives assignments from the SAT solver and propagates information to the Theory solvers. Internally, it works with

an E-graph structure. An assignment from SAT solver is processed by a congruence closure core, which is used to refine an E-graph. When nodes in the E-graph are merged, a new equality is propagated into the Theory solver. And also, a Theory solver may produce an equality, which is propagated into the congruence closure core. In some cases, this module may generate fresh atoms.

Theory solvers are implemented as a set of modules, each responsible for reasoning for a given theory. The solvers are divided into several categories: Basic theories, which are Boolean theories, Equality and Uninterpreted Functions and Arithmetic, which serves as basis for other theory solvers and Reducible theories, Hybrid theories and External theories. Boolean theories serve as the core for solving a theory of Bit-vectors using the technique of Bit-blasting. Equality and Uninterpreted functions theory is solved mainly in Congruence Closure Core. Reducible theories, such as Array and Datatypes theories, are reducible to base theories. Hybrid theories are theories that combine more solving in one theory. An example is the theory of strings, regular expression, and sequences, which combines equational solving with integer arithmetic.

During solving, a new clause may be added to the formula, which may contain fresh variables. The Z3 solver uses a garbage collector which removes unnecessary clauses whenever a searched branch is closed (e.g. by detecting conflict and bactracking). Along with clauses, terms and atoms which were introduced are no longer used are deleted too. Some of the Boolean atoms have no impact on the result. Such atoms are then marked as *don't care* and for some theories, these are ignored by Z3.

An E-graph used by the Congruence Closure Core is also used to remove quantifiers, where a quantified variable is instantiated in the E-graph and efficiently matched, using the *E-matching engine*.

In addition, a model is generated during the solving, which is used as part of the output. A value is assigned to constants, and for predicate and function symbols partial graphs are generated.

2.2.7 SMT-LIB Standard

SMT-LIB standard defines a language for the description of SMT problems, standard theories of SMT, and collects a set of benchmarks created over these theories. It was founded and maintained by the Satisfiability Modulo Theories Library (SMT-LIB), which is an „international initiative aimed at facilitating research and development in Satisfiability Modulo Theories (SMT)“ [3]. A full description of the standard can be found in literature [4], here a short summary is provided.

The language of the SMT-LIB standard is a first-order sorted language. A *sorted* means typed, where for each term a unique sort is assigned. An example of sort is `Int, Bool` etc. Each function has a *rank*, which is a sequence of nonempty sorts, where the input parameters are the first n sorts in a sequence for $n \geq 0$, and the last sort in the rank is the return type of the function. A *term* may contain a constant, variable, function symbol, quantifier (`foreach, exists`) and one of two keywords *let* (renaming symbols in term) and `match` (pattern matching).

The language is in strict prefix notation, where each term or given part of the term is enclosed in brackets, i.e. (`forall` (x `Int`) (`and` (`=` (`div` x 2) 0) y) refers to a formula, where x has a sort `Int`, y is free in formula, `forall` is a quantifier, `and` is Boolean operator, `=` is identity operator and `div` is integer division operator.

```
(theory Example
  :sorts ( (Num 0) )
  :funs  ( (NUMERAL Num)
           (Eq Num Num Bool)
           (+ Num Num Num)
           (f Num Num Num)       ; some function
           (f Num Num Num Num)   ; also overriding is allowed
         )
  :definition
  "An Example theory with equivalency and addition."
  :values
  "Each value can be a positive integer."
```

Figure 2.3: An example of theory, which defines one sort and four functions.

The SMT-LIB standard provides a way to define a theory. This definition is a descriptive file that provides the sorts of given theory and its values, definition of functions, and description of the theory. Each part of the definition starts with an attribute. An example of a small theory can be seen in Figure 2.3. The definition of theory defines only an interface of functions, not a definition of body of a function. The behavior of each function is defined in the theory solver. Standard theories are described in Section 2.2.2, where theory of equality is not part of the standard. These theories are used as building blocks for standard logics.

The main part of the SMT-LIB standard is the script that describes the input formula, which is then processed by the solver. The script is composed of a header, where information about script, used logic, and some other description can be found, and a body, where definitions, formulas, and commands are written. The body of the script can have the following structure: *function and constant declaration and definition*, *formula part* and *exit*.

The function and constant declarations are based on first-order logic. A constant is a nulary function symbol. So, the declaration of constant `(declare-const x Int)` is equivalent to the declaration of a function of rank with single sort `(declare-fun x () Int)`. In the *formula part* of the script, each term starts with `assert` command. There can be a sequence of assertions. To invoke the solver, a `check-sat` command is used, and the model can be requested with the command `get-model`. In a script, an assertion stack can be used with commands `push` and `pop`, where checking satisfiability of given formula part can be found and then these assertions are removed from formula after pop. The script ends with the command `exit`.

The information provided in this section should help with the orientation in STM-LIB files. The language defined by standard is used in a variety of different solvers. All the benchmarks collected by STM-LIB are provided in the SMT-LIB format.

Chapter 3

String Solving

This chapter provides a description of the theory of strings and regular expressions. In solvers, the theory of string is implemented as a part of the theory of sequences. The theory of sequences, in general, is not constrained only to sequences of chars but, in general, to sequences of any available sorts. In comparison, the theory of strings works only with sequences of characters, precisely sequences of unicode symbols. First, a definition of the theory and its signature is provided. Then, this chapter aims to describe the approaches used in solvers CVC5 and Z3.

3.1 Theory of strings and regular expressions

Theory of strings T_S contains three sorts: *Str* is a set of sequences over a finite alphabet, `Int` is an integer, and `Lan` is a regular language. The alphabet defined by the STM-LIB standard contains all UTF-8 characters, and the standard also defines a set of core and additional functions, which are often referred to as extended functions [3, 20]. The notation used in this work is modified in comparison to the standard. The functions are defined in the SMT-LIB standard notation (last sort of rank is the return type of the function).

In the following sections, s, t, with a possible subscript, denotes a string, l denotes a constant string literal, i, j, m, n denotes integers, x, y denotes variables, and ε denotes an empty string. The core functions are `con Str Str Str` is the concatenation of strings, for simplicity, may be used as (`con` $s_1 \dots s_n$), for $n \geq 2$ as the concatenation of n strings. The function symbol `len Str Int` is the number of characters in the string, in this work it can also be written as $|s|$ with the same meaning. A function symbol `<` is used for the lexicographical ordering of words as `< Str Str Bool`. A function symbol `toRe Str Lan` is the mapping of word s to the regular language $\{s\}$, a function symbol `inRe Str Lan Bool` returns if given string is member of regular languge, function symbols `reCon Lan Lan Lan`, `union Lan Lan Lan`, `inter Lan Lan Lan` and `star Lan Lan` is in order concatenation, union, intersection and Kleene star operator over regular language. In regular language, in SMT-LIB syntax, the following constants are also defined: `re.none Lan` represents empty regular language, `re.allchar Lan` is a representation of alphabet, a symbol Σ will denote this constant and `re.all Lan` represents the regular language of all words including ε, which will be denoted by symbol Σ^*.

Only a subset of extended functions is defined. A function symbol `substr Str Int Int Str` returns a substring (that is, `substr`(s, i, j) in string s at index i takes a substring of length at most j); the result may be an empty string if $i \geq |s|$ or $j < 0$. A function symbol

`contains Str Str Bool` is a containment of a string in a string (that is, `contains`(s,t) returns `true` iff t is a substring of s). A function symbol `indexOf Str Str Int Int` is the index of the first occurrence of the first characer of the string beginning search at a given index in string (that is, $textttindexOf(s,t,i)$ returns the index of the beginning of string t if it appears in string s at index i or later), returns -1 if $i \geq |s|$, t is not in s, or returns i if t is an empty string and $0 \leq i \leq |s|$. A function symbol `replace Str Str Str Str` denotes a replacement for a substring in string (that is, `replace`$(s\ t\ t')$ in string s replace the first occurrence of t with t'), the result may be unchanged string, when t is not a substring of s. Also, when t is an empty string, the resultant string will have the form (`con` t' s). A function symbol `strToInt Str Int` converts a character string in decimal notation to an integer or returns -1 when the string contains a non-digit character (including $-$, i.e. -575 returns -1 meaning error, not a negative integer). A function symbol `intToStr Int Str` converts non-negative integer into its string representation or returns ε otherwise.

3.2 CVC5 methods

The CVC5 solver approaches are incomplete and are not guaranteed to terminate, but in practical usage produce good results. The solver itself is a combination of linear arithmetic solver and modified congruence-closure-based solver for solving EUF, extended with string-solving specific and regular languages specific derivative rules. [13]

This section provides a summary of the methods used for string solving in CVC5. In the end of this section, some implementations details about CVC5 string solver are provided.

3.2.1 Basic calculus

The core calculus of the solver works with only the core functions of string theory and is described in the literature [13]. The following is a summary of it, and the description is not complete; examples are provided. The solver is built on the congruence closure algorithm, which generates a set defined as follows.

Definition 3.1 *Let S be a set of string constraints, and let $\mathcal{T}(S)$ be the set of all terms. A congruence closure $\mathcal{C}(S)$ is the set*

$$\mathcal{C}(S) = \{s = t \mid s,t \in \mathcal{T}(S), S \vDash s = t\} \cup \{l_1 \neq l_2 \mid l_1, l_1 \textit{ distinct string constants}\} \cup \{s \neq t \mid s,t \in \mathcal{T}(S), s' \neq t' \in S, S \vDash s = s' \wedge t = t', \textit{ for some } s', t'\}$$

An equivalence relation E_S induced by $\mathcal{C}(S)$ is the relation over $\mathcal{T}(S)$, where the terms $t, s \in \mathcal{T}(S)$ are equivalent iff $t = s \in \mathcal{C}(S)$ and such terms create an equivalency class denoted $[t]_S$.

$$\begin{aligned}
\mathrm{con}(s, c_0 \dots c_i, c_{i+1} \dots c_n, u) &\to \mathrm{con}(s, c_0 \dots c_n, u) & \mathrm{len}(c_0 \dots c_n) &\to n+1 \\
\mathrm{con}(s, \mathrm{con}(t_0, \dots t_n), u) &\to \mathrm{con}(s, t_0, \dots t_n, u) & \mathrm{con}(s, \varepsilon, u) &\to \mathrm{con}(s, u) \\
\mathrm{len}(\mathrm{con}(s_0, \dots, s_n)) &\to \mathrm{len}(s_0) + \dots + \mathrm{len}(s_n) & \mathrm{con}(s) &\to s \\
& & \mathrm{con}() &\to \varepsilon
\end{aligned}$$

Figure 3.1: A set of normalization rules.

$$\text{S-Split}\frac{x, y \in \mathcal{V}(S) \quad x = y, x \neq y \notin \mathcal{C}(S)}{S := S \cup \{x = y\} \quad \| \quad S := S \cup \{x \neq y\}} \qquad \text{N-Form2}\frac{[x] \notin \mathcal{D}(N) \quad [x] \subseteq C \cup \mathcal{V}(S)}{N := N \cup \{[x] \mapsto (x)\}}$$

$$\text{A-Conflict}\frac{A \vDash_{LIA} \bot}{\texttt{unsat}} \qquad \text{S-Conflict}\frac{s = t \in \mathcal{C}(S) \quad s \neq t \in \mathcal{C}(S)}{\texttt{unsat}}$$

$$\text{Reset}\frac{}{F := \varnothing \; N := \varnothing \; B := \varnothing}$$

Figure 3.2: Example of derivation rules from [13], to demonstrate the building of the context and flow of the computation.

Definition 3.2 *A calculus configuration is a special configuration* **`unsat`** *or a septuple* $\langle S, A, R, F, N, C, B\rangle$ *where*

S, A, R *are sets of string, arithmetic and regular language constraints,*

F *is a set of pairs* $s \mapsto (a_0, \ldots a_n)$ *where* $s \in \mathcal{T}(S)$ *and* $(a_0, \ldots a_n)$ *is a tuple of atomic string terms,*

N *is a set of pairs* $e \mapsto (a_0, \ldots a_m)$ *where* e *is an equivalence class of* E_S *and* $(a_0, \ldots a_m)$ *is a tuple of atomic string terms,*

C *is a set of terms of sort* **`Str`**,

B *is a set of buckets where each bucket is a set of equivalency classes of* E_S.

The sets S, A, R are at first the input problem and grow as new terms are introduced during solving. The set F stores the term in intermediate form (also called flat form), from which the term is computed in normal form. The normal form terms are stored in N and C contains terms, which should not be transformed into intermediate form to prevent computation cycle. The normal form calculation is illustrated in Figure 3.1. Note that the `len` function over constant sequence c_i is just a number of characters (indexing from 0 causes +1).

Definition 3.3 *The derivation tree for the calculus is a tree, where each node is a configuration and each non-root node is obtained by applying one of the derivation rules on its parent. The rood node is called the initial configuration. A branch of the derivation tree is called closed if it ends with* **`unsat`**. *A tree is called closed when every branch is closed.*

For summary purposes, it is necessary to provide a simplified definition of two new symbols. A set N is the partial map from E_S to normalized tuples of atomic terms. The domain $\mathcal{D}(N)$ of the partial map N is the set $\{e \mid e \mapsto (a_0, \ldots, a_n) \in N \text{ for some } (a_0, \ldots, a_n)\}$. A set $\mathcal{V}(S)$ is the set of free variables in S.

The calculus is based on the set of *derivative rules* in *guarded assignment form*. A rule is applied in a configuration K, if all the rule premises are valid for K. The symbol $\|$ separates two conclusions of a given rule, which simulates a nondeterministic decision of the procedure.

In Figure 3.2, five rules can be seen. First, each rule produces a new context by updating one or more of its sets or producing a special context `unsat`. Rule *S-Split* is simple rule, which demonstrates nondeterministic decision, where two branches are created. Rule *N-Form*

shows one of the forms of creating a normal form, which is computed from the equivalency class, and a mapping is assigned created for each normal form formula. Two rules, *A-Conflict* and *S-Conflict* are the only two rules that produce an `unsat`. The *S-Conflict* produces `unsat` when there are conflicting equivalences in set S. The *A-Conflict* rule generates `unsat` when the linear arithmetic solver entails $\bot$, in other words, the arithmetic solver found some conflict. The *Reset* rule here is a bit different from DPLL(T) `Reset`, because is meant to be applied each time, when a set S is updated. It is the only rule which erases some parts of the context, which leads to the fact that the sets S, A, R, C only grow during the computation in a given branch (the set C prevents the algorithm from looping). Also, *Reset* is applied because after updating the set S, flat form and normal form sets may need to be updated.

Earlier, a definition of closed tree was provided; such a tree induces an unsatisfiability of the formula. In addition, the configuration may enter a *saturated configuration.*

Definition 3.4 *A saturated configuration is configuration* $\langle S, A, R, N, F, C, B\rangle$ *where*

i) N *is total map over* E_S*,*

ii) B *is a partition of* E_S*,*

iii) any derivation rule that applies to it except for Reset leaves the configuration unchanged modulo renaming of Skolem variables.

The authors declare the correctness of this approach to string solving. If all derivation trees with the root $\langle S_0, A_0, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing\rangle$ are closed, then if $S_0 \cup A_0$ is unsatisfiable in the theory of strings, then a solver is a refutation sound. Also, solution soundness proposes that, when there exists a branch with saturated configuration from root $\langle S_0, A_0, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing\rangle$, then $S_0 \cup A_0$ is satisfiable in the theory of strings.

3.2.2 Extended function simplification

The CVC5 solver handles extended function by simplifying it into the basic string solving function. The procedure of simplification extends the basic calculus with processing of extended string functions. The complete description can be found in the literature [20]. Here, a short summary is provided.

Several changes were made to the calculus configuration. It was extended with the set G, which is a set of formulas on extended string theory. S is now the internal state of the solver in the form $S = (E, X, F, N)$, where F, N has the same meaning as in basic calculus, a set E is a set of basic string equalities and X is a set of equalities in the form $x = t$, where x is a string variable and t is a flat extended function term. The flat term is the term of the form $f(x_0, \ldots, x_n)$, where $x_0, \ldots, x_n$ are variables. The $\lfloor\varphi\rfloor$ denotes an equisatifiable (purified) form of formula φ and $\varphi\{x \mapsto y\}$ denotes a substitution of x by y in φ.

The basic calculus is extended with rules for handling extended functions. The decidability of an extended string function is an open question, so the algorithm is not guaranteed to terminate. However, as in basic calculus, the properties of refutation and solution soundness hold.

The first approach to simplification of extended function is to simplify formula over extended string theory to possibly quantified formula over basic string theory (theory without extended functions). The rule for this reduction *Ext-Expand* can be seen in Figure 3.3. A formula $x = t$ is simplified into an equisatisfiable formula $[\![x = t]\!]$, therefore, when

$$\text{Ext-Expand}\frac{x = t \in X}{G := G \cup \{\lfloor [\![x = t]\!] \rfloor\} \qquad X := X \setminus \{x = t\}} \quad \text{where}$$

$$\begin{aligned}
[\![x = \texttt{substr}(y, n, m)]\!] = \texttt{ite}(&0 \leq n < len\ y \wedge 0 < m,\\
&y = \texttt{con}(z_1, x, z_2) \wedge \texttt{len}\ z_1 = n \wedge \texttt{len}\ z_2 = \texttt{len}\ y - m,\\
&x = \varepsilon)\\
[\![x = \texttt{contains}(y, z)]\!] = (x \neq \top) \Leftrightarrow \forall k.0 \leq k \leq \texttt{len}\ y - \texttt{len}\ z &\Rightarrow \texttt{substr}(y, k, \texttt{len}\ z) \neq z\\
[\![x = \texttt{indexOf}(y, z, n)]\!] = \texttt{ite}(&0 \leq n \wedge z \neq \varepsilon \wedge \texttt{contains}(y', z),\\
&\texttt{substr}(y', x', \texttt{len}\ z) = z \wedge\\
&\neg\texttt{contains}(\texttt{substr}(y', 0, x' + \texttt{len}\ z - 1), z),\\
&x = -1)\\
&\text{with } y' = \texttt{substr}(y, n, \texttt{len}\ y - n) \text{ and } x' = x - n\\
[\![x = \texttt{replace}(y, z, w)]\!] = \texttt{ite}(&\texttt{contains}(x, z) \wedge z \neq \varepsilon,\\
&x = \texttt{con}(z_1, w, z_2) \wedge y = \texttt{con}(z_1, z, z_2) \wedge\\
&\texttt{indexOf}(y, z, 0) = \texttt{len}\ z_1,\\
&x = y)
\end{aligned}$$

Figure 3.3: Simplifying rules for the extended functions.

$$\text{B-Val}\frac{t : \texttt{Int} \quad n \text{ is a numeral}}{A := A \cup \{t \leq n\} \quad \| \quad A := A \cup \{t > n\}} \qquad \text{B-Inst}\frac{\begin{array}{r} G = G' \cup \{\varphi[\forall k.0 \leq k \leq t \Rightarrow \psi]\} \\ A \vDash_{\text{LIA}} t \leq n \text{ for some numeral } n \end{array}}{G := G' \cup \{\lfloor \varphi[\wedge_{i=0}^{n} \psi\{k \mapsto i\}] \rfloor\}}$$

Figure 3.4: Elimination of bounded quantifier.

$[\![x = t]\!]$ is satisfiable, $x = t$ is also satisfiable. Also, the set X is possibly empty in saturated configuration, and hence all the extended functions were simplified.

Looping over the *Ext-Expand* rule is not possible, because the expansion of each function contains only preceding functions in order `substr, contains, indexOf, replace` (i.e, `indexOf` expands to `contains` and `substr`, `substr` is expanded to basic functions); therefore, no recursion occurs.

The unwanted result of this simplification is the possible introduction of a new quantified formula. This formula is of the form $\forall k.0 \leq k \leq t \Rightarrow \varphi$ is called *integer bounded*. The integer bounded formula allows the quantifier to be eliminated by introducing two other rules from Figure 3.4: *B-Val* and *B-Inst*. The *B-Val* rule splits the search for guessing the upper bound, whereas *V-Inst* rule takes the formula φ which contains a subformula $\forall k.0 \leq k \leq t \Rightarrow \psi$ and, if A entails a concrete upper bound, replaces it with a conjunction. Due to bounds, the conjunction is finite and these two rules are sufficient for eliminating a quantifier.

The described approach may generate unwanted complexity of the formula and may be ineffective in space. An improvement for this method is a context-dependent simplification of the formulas. Such simplification may avoid some of the rewriting of extended functions when an algorithm can infer a result of the function. For example, result $\texttt{replace}(x, y, z) = x$, when y is not in x. In the original article are described three rules of the calculus that

$$
\begin{aligned}
\texttt{contains}(l_1, l_2) &\to \top && \text{if } l_1 \text{ contains } l_2 \\
\texttt{contains}(l_1, l_2) &\to \bot && \text{if } l_1 \text{ does not contains } l_2 \\
\texttt{contains}(l_1, \texttt{con}(l_2, t_0, \ldots, t_n)) &\to \bot && l_1 \text{ does not contains } l_2 \\
\texttt{contains}(\texttt{con}(l_1, t_0, \ldots, t_n), l_2) &\to \top && l_1 \text{ contains } l_2 \\
\texttt{contains}(\texttt{con}(x, t_0, \ldots, t_n), s) &\to \top && \text{if } \texttt{contains}(\texttt{con}(t_0, \ldots, t_n), s) \to^* \top
\end{aligned}
$$

Figure 3.5: An example of simplification rules for `contains`.

handle such cases. Explanations of these rules are out of the scope of this work. In short, these rules, based on the equivalencies of elements of the sets X and E, add a simplified formula t to the set G based on its properties. In addition, a substitution is made for the variables of t. The simplified formula is obtained by applying the set of rules over extended functions. An example of these rules for function `contains` are provided in Figure 3.5.

The ability to perform such simplification depends on choosing the right substitution over the variables of the term t. Two possible heuristics for choosing the appropriate substitution are used. The first is to choose a substitution $\{y \mapsto s\}$ for each variable y in the term t, to some representative of the equivalency class $[y]$. This substitution is called *representative substitution*, which is computationally easy and provides satisfactory results in practice.

The second heuristic is called *normal form substitution* where each free variable y in term t is substituted with a simplified $\texttt{con}(a_0, \ldots, a_n)$ (in terms where $\texttt{con}(a_0, \ldots, a_n)$ cannot be simplified further), where $y \in [y']$, y' is representative of a given equivalency class, and $y' \mapsto (a_0, \ldots, a_n) \in N$. The advantage of this heuristic is that it provides all inferred information about t, giving the most accurate context to apply the simplification rules.

3.2.3 Arithmetic-Based Simplification

Another system used for simplification of input formulas is arithmetic-based inference of length of variables in the theory of strings. For example, $\texttt{substr}(s, m, n)$ can be simplified to an empty string when it can be shown that $m > |s|$ holds. This section provides a definition of term in polynomial form, a schema, which infers over these terms and an example of simplification consequent to the inference schema. This section provides a summary of inference system used in CVC5 [18].

The scope of the arithmetic-based inference system aims to prove that formulas of the form $u \geq 0$ are valid in the theory of strings, where u is a term of `Int` sort. The system contains rules that derive judgements in the form $\vdash u \geq 0$ and a strategy to guide the application of these rules. This system is incomplete, due to complexity of reasoning over the theory of integer arithmetic (see Section 2.2.2), however, it is sound in the sense that $\vDash_{T_S} u \geq v$ whenever $\vdash u \geq 0$ is derivable in the system.

Definition 3.5 *An arithmetic term u is in polynomial form if $u = m_1 \cdot u_1 + \ldots + m_n \cdot u_n + m$, where $m_1, \ldots, m_n$ are nonzero integer constants, m is an integer constant and each $u_1, \ldots, u_n$ is a unique term and one of the following:*

1. *an integer variable,*
2. *an application of length to a string variable, e.g. $|x|$,*

$$\frac{u \rightarrow_U^* n \quad n \geq 0}{u \geq 0} \qquad \text{where}$$

$$
\begin{aligned}
|t| &\rightarrow_U 0 \\
|\texttt{substr}(t, m, n)| &\rightarrow_U \begin{cases} n & \text{if } \vdash m \geq 0 \text{ and } \vdash |t| \geq m + n \\ |t| - m & \text{if } \vdash m \geq 0 \text{ and } \vdash m + n \geq |t| \end{cases} \\
|\texttt{replace}(s, t, t')| &\rightarrow_U \begin{cases} |s| & \text{if } \vdash |t'| \geq |s| \text{ or } \vdash |r| \geq |t| \\ |t| - |s| & \end{cases} \\
|\texttt{intToStr}(m)| &\rightarrow_U 1 \qquad \text{if} \vdash v \geq 0 \\
\texttt{indexOf}(s, t, v) &\rightarrow_U -1 \\
\texttt{strToInt}(s) &\rightarrow_U -1 \\
c \cdot m + i' &\rightarrow_U c \cdot n + i' \text{ if } m \rightarrow_U n \text{ and } c > 0 \text{ or } m \rightarrow_O n \text{ and } m < 0
\end{aligned}
$$

$$
\begin{aligned}
|\texttt{substr}(s, m, n)| &\rightarrow_O n \qquad \text{if } \vdash n \geq 0 \\
|\texttt{substr}(s, m, n)| &\rightarrow_O \begin{cases} |s| - m & \text{if } \vdash |t| \geq m \\ |s| & \end{cases} \\
|\texttt{replace}(s, t, t')| &\rightarrow_O \begin{cases} |t| & \text{if} \vdash |s| \geq |r| \\ |t| + |r| & \end{cases} \\
|\texttt{intToStr}(m)| &\rightarrow_O \begin{cases} m & \text{if } \vdash m > 0 \\ m + 1 & \text{if } \vdash m \geq 0 \end{cases} \\
\texttt{indexof}(s, t, m) &\rightarrow_O \begin{cases} |s| - |t| & \text{if } |s| \geq |t| \\ |s| & \end{cases} \\
c \cdot m + i' &\rightarrow_O c \cdot n + i' \text{ if } m \rightarrow_O n \text{ and } c > 0 \text{ or } m \rightarrow_U n \text{ and } m < 0
\end{aligned}
$$

Figure 3.6: Under and over-approximation rules for arithmetic inference.

3. *an application of length to an extended function, e.g.* $|\texttt{substr}(s, m, n)|$, *or*

4. *an application of an extended function of integer type, e.g.* $\texttt{indexOf}(s, t, m)$.

In addition, the term u is considered in a simplified form, where all constant string literals $|l|$ of length n are substituted with n, the terms of $|\texttt{con}(t_0, \ldots, t_n|$ are rewritten to form $|t_0| + \ldots |t_n|$, and arithmetic simplification is used whenever possible (e.g. $7 \cdot |x| - 2 \cdot |x|$ is rewritten to $5 \cdot |x|$, etc.).

The inference system uses rules for the under-approximation denoted $\rightarrow_U$ and the over-approximation denoted $\rightarrow_O$. The $m \rightarrow_{\{U,O\}}^* n$ denotes that m is rewritten to n in zero or more steps in a given system. If $m \rightarrow_U^* n$, then $\vDash_T n \geq m$ holds. Also, if $m \rightarrow_O^* n$, then $\vDash_T m \geq n$ holds. For the given term u in polynomial form, it can be shown that $u \geq 0$ in all models of T_S, with the inference rule shown in Figure 3.6.

Some of the terms may be rewritten in several ways. Due to that, an approximation strategy needs to be provided. The goal of the strategy is to find non-negative integer n, such

STR-ARITH-APPROX(u), where $u = u_x + u_l + u_s + c$ and:

$$\begin{aligned} u_x &= c_1^y \cdot y_1 + \ldots + c_p^y \cdot y_p \\ u_l &= c_1^l \cdot |x_1| + \ldots + c_q^l \cdot |x_q| \\ u_s &= c_1^m \cdot m_1 + \ldots + c_r^m \cdot m_r \end{aligned}$$

for variables $x_1, \ldots, x_q$, $y_1, \ldots, y_p$ and extended terms $m_1, \ldots, m_r$:

1. If $r > 0$, choose a m_i and m_i' that maximize the following criteria (in descending order), where $u' = (u\{c_i^m \cdot m_i \mapsto c_i^m \cdot m_i'\})$ and u' is simplified:
   (a) (Soundness) $m_i \to_U m_i'$ if $c_i^m > 0$ and $m_i \to_O m_i'$ if $c_i^m > 0$,
   (b) (Avoid new terms) minimizes the size of $\texttt{negcoeff}(u') \setminus \texttt{negcoeff}(u)$,
   (c) (Cancel existing terms) maximizes the size of $\texttt{negcoeff}(u) \setminus \texttt{negcoeff}(u')$.

   Return $u \to_U u'$.
2. If $q > 0$ and $c_j^l > 0$ for some j, return $u \to_U (u\{c_j^l \cdot |x_j| \mapsto 0\})$.

Figure 3.7: An efficient strategy to achieve arithmetic entailment in T_S.

that $u \to_U^* n$, which provides sufficient proof that $\vDash_T u \geq 0$ holds. The provided strategy splits u into three parts $u = u_x + u_l + u_s + c$, where u_x is a sum of integer variables, u_l is a sum of length of string variables, u_s is a sum of extended term variables (e.g. $|\texttt{intToStr}(m)|$, $\texttt{indexOf}(s, t, v)$, etc.) and c is integer constant. The approximation rules are applied in a way that eliminates as many members of the polynomial as possible. Thus, the strategy works especially with the u_s part of u.

In Figure 3.7 the abstract algorithm for a given strategy is demonstrated. The symbol $\texttt{negcoeff}(u)$ returns a set of integer terms of u that has the integer part $c < 0$ (e.g. $\texttt{negcoeff}(|x_1| + -5 \cdot |\texttt{intToStr}(s)|) = \{|\texttt{intToStr}(s)|\}$).

If the part u_s has non-zero terms, the algorithm tries to apply substitution on each term, such that for soundness it chooses an over-approximation for negative coefficients and an under-approximation for positive ones. In general, the strategy tends to avoid introducing new terms with negative coefficients, to prove that $\vdash u \geq 0$ holds. A new term is introduced when it is with positive coefficient or when it leads to eliminating already existing negative term. The second part of the algorithm removes all length terms with positive coefficients, only when the u_s part is empty because these terms can help eliminate other negative terms. After each substitution, the term u is simplified with standard arithmetic rules.

This inference system is used to investigate possible strings' sizes and the values of integer terms. These entailed values can be used to guide a set of simplification rules. In Figure 3.8 an example of possible simplification rules is provided.

Provided simplifications are based on the given side condition. For example, the first rule shows that two terms cannot be equivalent when it can be inferred that the size of one string is strictly larger than the size of the other one. Also note that there are two rules for extended functions (`indexOf`, `contains`), where the extended functions are eliminated.

$$
\begin{array}{rll}
s = t \to \bot & & \text{if } \vdash |s| \geq |t| + 1 \\
s = \texttt{con}(t, u, v) \to s = \texttt{con}(t, v) \wedge u = \varepsilon & & \text{if } \vdash |t| + |v| \geq |s| \\
\texttt{substr}(s, m, n) \to \varepsilon & & \text{if } \vdash 0 > m \vee m > |s| \vee 0 \geq n \\
\texttt{substr}(\texttt{con}(s, t), m, n) \to \texttt{substr}(s, m, n) & & \text{if } \vdash |s| < m + n \\
\texttt{indexOf}(s, t, m) \to \texttt{ite}(\texttt{substr}(s, m, |s|) = t, m, -1) & & \text{if } \vdash m + |t| \geq |s| \\
\texttt{contains}(s, t) \to s = t & & \text{if } \vdash |t| \geq |s|
\end{array}
$$

Figure 3.8: Example of simplification rules that using arithmetic entailment system.

3.2.4 Other heuristics

In previous sections some of the approaches used in CVC5 were provided. Note that the approaches are mainly focused on simplifying the input formula. Extended functions are eliminated whenever possible in the system. In addition, a short summary of other approaches is provided in this section.

A *containment-based simplification* applies simplification rules according to, if a string contains other string. A set of inference rules about containment is provided along with a set of simplifying rule. The inference schema is based on syntactical inference, so it can be done statically. The inference rules show that if a string term s, t or literals l_1, l_2 and its combination contain each other, a conclusion about the terms can be drawn. For example, when l_1 contains l_2, or l_2- is a prefix of l_1, then $\texttt{con}(l_1, s)$ also contains l_2. Some of the judgements can be used along with the arithmetic system. [18]

A *multiset reasoning* uses the abstraction over strings, where a string is seen as a multiset (that is, each element can be contained more than once in the set). Reasoning over such abstraction is not based on comparison of strings, symbol by symbol, but on equality of number of characters in the string. Let M, N be two multiset abstractions such that $M = \{\texttt{e},\texttt{w},\texttt{w},\texttt{w}, x\}$ and $N = \{\texttt{w},\texttt{w},\texttt{w}, x\}$, then it can be shown that $M \setminus N = \{\texttt{e}\}$ and $N \setminus M = \{\}$, so that one of the abstractions contains at least one `e` more, than the other. Also, these abstractions are an over-approximation of a given term, so showing of these set differences is sufficient to reason about strings. [18]

A *witness sharing* is a technique used for effective elimination of quantifiers. In CVC5 exists a set of rules that splits cases for equivalency, eliminates the extended function, and eliminates some RE functions, and as a side effect of this elimination, an existential quantifier is introduced into the formula. The classical approach to eliminate existential quantifiers is by *Skolemization*, which introduce a fresh variable into formula. Introducing a new variable for each quantifier in the formula can lead to unnecessary increase in complexity. Thus, a witness term is introduced instead of instantiating a quantifier with a fresh variable. This term can lead to further simplification, as it can appear in more parts of the formula. As an example, the term $x \cdot x' = c \cdot y'$ can be used, where c is a single constant character, and if $|x| > 1$ it can be derived in the form $\exists k_1 . x = c \cdot k_1 \wedge k_1 \cdot x' = y'$, so x has a form of character and some suffix k_1. Here, a k_1 can be instantiated with a witness term in form $\texttt{substr}(x, 1, |x|)$ denoting a substring of x without the first character, leading to the term $x = c \cdot \texttt{substr}(x, 1, |x|) \wedge \texttt{substr}(x, 1, |x|) \cdot x' = y'$. But this also leads to an increase in complexity as it introduces a new extended function into the formula. Instead, an instantiation with the witness variable is used, and each of these variables has an associated

witness term. Whenever a quantifier is eliminated, before introduction of a new variable, it is first checked if the variable with the same witness term does not already exist, and if it does, the existing variable is used. [19]

For regular expressions, there are more approaches. One of them is to derive regular expressions using derivative rules. The second method is to reduce them to the extended string functions. The advantage of this approach is that the constraints can be processed only by a string solver since regular expressions can have its own solver. Thus, it leads to earlier inconsistency detection. As an example of the reduction of regular expressions into string constraints, only one rule will be provided, which reduces the term $x \in \texttt{reCon}(R_1, \texttt{toRe}(y), R_2)$, where x, y are string variables and R_1, R_2 are regular languages. The rule has a following form:

$$x \in \texttt{reCon}(R_1, \texttt{toRe}(y), R_2) \rightarrow \quad \exists i.0 \geq i < |x| - |y| \wedge \texttt{substr}(x, 0, i) \in R_1 \wedge \\ \texttt{substr}(x, i, |y|) = y \wedge \texttt{substr}(x, i + |y|, |s|) \in R_2$$

In short, this rule says that x is composed of a prefix of length of i, which is in R_1, a variable y, and a suffix that is in R_2. [19]

3.2.5 Implementation details

Based on previous sections, it is clear that the major part of the effective solver is based on the right simplification techniques. Although these descriptions of the techniques and the literature provide some level of implementation detail, the real implementation has to be further investigated. Here, a brief structure of the CVC5 string solver is provided. The code can be distributed under modified BSD license and the source code is available on GitHub [1]. This section aims only at the description of the string solver and its parts, not at a full description of the CVC5 solver. Also, information about implementation was obtained by exploring, thus it can be misinterpreted or incomplete.

The terms in the solver are stored in the class `Node` (an instance of `NodeTemplate`), each node has a value and it is possible to get the children of the node, hence the arrangements of the nodes form a tree structure. The `Node` can contain a variable, constant, or function, where the operands of the function are children of the node. `NodeManager` is responsible for creating nodes and providing some operations on them.

The string solver and its components can be found in the source structure in the directory `src/theory/strings`. The core part of the solver is the class `TheoryStrings`, that can be found in `theory_strings.{cpp,h}` files. This class holds an instance of all components of string solving. It implements a subclass `NotifyClass`, which is responsible for notifying the equality solver about new events in the solver, like a new literal assignment with boolean value.

The `TheoryStrings` class uses the class `ExtfSolver`, which implements the Extended Function Simplification from Section 3.2.2 and can be found in files `extf_solver.{cpp,h}`.

As proposed earlier, one of the most important parts of the solving is simplification of the input formula. Based on observations, the simplification is implemented in files `sequences_rewriter.{cpp,h}`, where most of the simplification rules provided can be found. The class `SequencesRewriter` is implemented in these files. A method `postRewrite` serves as an entry point from which rewrites are called for all functions. For example, in the method `rewriteContains` all the simplifying rules provided in Figure 3.5 can be found.

The Arithmetic-Based Simplification from Section 3.2.3 is implemented in class `ArithEntail` in files `arith_entail.{cpp,h}`. The basic inference schema, which is described

in Figure 3.6 is implemented in the method `checkApprox`. In the method there is a map `mApprox`, which, for each term (`Node`), holds a vector of all possible approximations of the term. From these vectors of approximations the most suitable one is chosen and applied to computation. In this step, the described strategy is applied.

In the files `strings_entail.{cpp,h}` is implemented the containment-based simplification and its rules for inference of containment. Also, a method which implements multiset reasoning is implemented here.

3.3 Z3 methods

There are many sources to study Z3 and the methods that this solver uses. The Z3 GitHub repository provides a page with publications about Z3 [9], and a description of the internals of Z3 is also provided [7, 8]. Although these sources and complex descriptions are available, the information about approaches used in the default Z3 string solver is short or completely missing. In addition, the Z3 code does not contain any summary of the provided string solving functionality [10]. However, a part of the solver, which provides handling of regular expressions, is described. This section provides a brief summary of regular expressions solving, which is taken from [22], and a description of the implementation, based on code exploration.

3.3.1 Symbolic regular expression derivatives

The common approach for solving a regular membership constraint in other solvers is based on converting regular expressions (RE) into finite automata and performing an operations corresponding to the input formula over them, or, an approach where the operation is propagated over RE and then reason over it with derivative rules. The construction of a finite automaton is typically expensive, as operations such as intersection and union causes an explosion in the number of states, or even a simple automaton corresponding to regex `.{`n`}`(n occurrences of any symbol) will have n states. In contrast, the second approach does not provide a blow-up in possible state space but does not provide an intersection and complement of RE and uses an over- and under-approximation of the regular language.

The *Symbolic derivatives* is based on theory of *Brzozowski* and *Antimirov* derivatives, which insist on knowing the first symbol of the regular expression. The symbolic evaluation in comparison, is able to provide derivatives without knowing that first symbol. That is possible by introducing a *transition regex*, that is, a (extended) regular expression extended by the *conditional regex* $\texttt{IF}(\varphi, r_1, r_2)$, where φ is a term and r_1, r_2 are transition regexes. The transition regex is a function that maps symbols to RE. As an example, only rules for the conditional regex are provided.

The first rule, which is the function application, is further used for computing derivatives. An example of application can be $\texttt{IF}(x = \texttt{a}, r_1, r_2)(\texttt{b})$, where `a, b` are symbols and the result of the application of the parameter will be $r_2(\texttt{b})$, which may be further applied, or if r_2 is a regular language, the parameter will be discarded. Note that the application of concatenation and negation does not affect the condition φ and only applies to expressions r_1, r_2.

$$\texttt{IF}(\varphi, r_1, r_2)(x) = \begin{cases} r_1(x) & \text{if } x \text{ is } \texttt{true} \text{ in } \varphi \\ r_2(x) & \text{otherwise} \end{cases}$$
$$\texttt{IF}(\varphi, r_1, r_2) \cdot R = \texttt{IF}(\varphi, r_1 \cdot R, r_2 \cdot R) \qquad R \text{ is RE}$$
$$\overline{\texttt{IF}(\varphi, r_1, r_2)} = \texttt{IF}(\varphi, \overline{r_1}, \overline{r_2})$$

Figure 3.9: Example of transformation of *conditional regex*. $r_1 \cdot r_2$ is RE concatenation and $\overline{r}$ is negation.

The *symbolic derivative* $\delta(R)$, where R is RE, is the transition regex with a set of defined rules. Some of the rules are

$$\delta(\varepsilon) = \delta(\bot) = \bot$$
$$\delta(\varphi) = \texttt{IF}(\varphi, \epsilon, \bot)$$
$$\vdots$$

Also, for the computation, the rule *lift* is used to propagate the conjunctions to the bottom level of the transition regex, and the conditionals are lifted.

An implementation of a regular expression solver maintains a graph G, that is, a directed graph, which contains information about the regexes. Each vertex represents an RE and each egde represents a derivation, which implies that the graph is acyclic. The graph abstraction is then used to compute the satisfiability of $s \in r$, where s is a string and r is a regular expression, iteratively removing a prefix symbol from s until $|s| = 0$ so it is satisfiable, or it can be shown that $s \in r$, where r cannot be further derived, leading to unsatisfiability.

3.3.2 Implementation details

The structure of the Z3 string solver is not documented. The general structure is based on the core equality solver, which sees the theory solver as modules. The description in this section is based on the exploration of code [10] and inferring the functionality of each component, so it can be misinterpreted and will be incomplete.

In the source tree, the string solver components appear in different directories, so it is unclear which of the modules are used together. The theory plugins can be found in `src/smt` directory.

The class `theory_seq`, implemented in files with the same name, is the native Z3 theory plugin for strings and sequences. This class inherits from class `Theory`, which contains the solver's context. The member of the `theory` class is the `context` class, implemented in the files `smt_context.{cpp,h}`, which contains a set of member variables and classes, which holds a required modules (e.g., rewriter of type `th_rewriter`, model generator, statistics, etc.). The `theory_seq` class implements a set of subclasses that are used in the solver for regular expressions. The subclasses serve either as containers for some data or have an overriden `operator()`, thus can be used as methods.

The class `theory_str` is implemented in the source and header of the same name and it is the main theory plugin for the Z3Str3 solver, therefore, it is not within the scope of this work.

Another part, which is implemented in the `smt` directory, is the class `seq_axioms`, which implements a set of operations on the input expression that simplifies it. This implementation is only a thin wrapper over the second implementation, which will be described later.

In the folder `seq/ast` are files `seq_decl_plugin.{cpp,h}`, where a set of helper classes is implemented, which manipulates sequences with an enum class, which defines *sorts* and `kinds` of nodes. Sorts are the sorts of SMT-LIB standard, kinds are types of nodes (e.g. concatenation, contains, length, etc.). The helper classes provide a set of methods used to get kinds and types of string terms (e.g., `is_char, are_equal` and creational methods `mk_sort, mk_concat`, etc.).

Finally, the rest can be find in directory `src/ast/rewriter`. In `seq_axioms.{cpp,h}` can be found the implementation of class `axioms`, which implements the set od theory strings axioms. This class is used by the class `seq_axioms` mentioned before. Many of the functions in the source file have a comment with description of the used axioms. Most of the comments are in the form of first-order logic formulas. The source code of `axioms` class is one of the most documented part of the string solver.

The class `eq_solver` from from files `eq_solver.{cpp,h}` implements the inference rule about equalities. Some of the methods have a comment on the rule used. The rules are written in guarded assignment form.

In the files `seq_skolem.{cpp,h}` a helper class is implemented, which helps to axiomatize operations on sequences.

And the last class, which implements the string solver, is `seq_rewriter`, which can be found in the files `seq_rewriter.{cpp,h}`. The importance of this class can be inferred from the size of the file, which has a bit more than 6000 lines, from which about 5100 are lines of code and 550 are comments (the only bigger file is file `polynomial.cpp`, which contains more than 6000 lines of code[2]). The class contains a set of methods with rules for simplifying the solved formula which returns a `br_status` – that is, a level of used rewrite (built-in rewrite), which may have status done, failed or rewrite with level 1-3 or rewrite full, which is unbounded. The failed rewrite means that there is no suitable rewrite for a given input. The rewriter methods typically first checks, if some part of input expression is constant and if it is constant, it computes a result of given expression (e.g. returns length of the string, for `contains` it returns bool, or the result concatenation, etc.). Some of these results can be done partially (e.g., simplification rules for concatenation in Figure 3.1). The entry point of simplification is method `mk_app`, which call `mk_app_core`, which calls other methods of this class.

In the `seq_rewriter` is also implemented a set of checker methods (comparison of characters, check of prefix, suffix, etc.) and also a few methods for arithmetic inference over lengths.

And one of the major parts of the rewriter is implementation of regular expression derivatives, which in coordination with `seq_regex` class is an implementation of the symbolic derivatives from Section 3.3.1. An entry point to the computation of derivatives serves the method `mk_re_derivative`. From this method a recursive algorithm starts, whose result is a derivative of a regular expression.

The inputs of the rewriter methods are of type *expr*, which inherits from class `ast`. This class is the implementation of the directed acyclic graph (DAG); thus, the AST is probably short for *abstract syntax tree*. Such an expression is considered only as the leaf of the tree,

[2]Counted with Visual Studio Code plugin VS Code Counter https://github.com/uctakeoff/vscode-counter.

but when the kind of node is a function, it is internally casted to the `app` class, from which parameters can be obtained.

Chapter 4

Measurements

The approach of comparing the efficiency of the solver is to run the solver on some set of benchmarks. Then, based on a given criterion, is the solver compared to other existing solvers. In the provided sources on used heuristics, typically is the new enhanced iteration of the solver compared to others according to the successfully solved benchmarks – more of the benchmarks were marked as `sat` or `unsat`, than timeouted. In addition, some of the comparisons are made according to the time of solving. The goal of this work is to analyze and compare the impact of the existing heuristic mainly on the latter criterion.

For clarity, the *original* or *full* solver is a solver obtained from official repositories without further modification. The *modified* has some of the heuristics disabled. The full version has the suffix `_full`, modified versions uses the suffix -<*name*>, where name is some short name for disabled heuristic.

In this chapter, the experiments performed and the results reached are described. First, a brief description of the benchmarks used and a description of the benchmarking tool used will be provided. Then a criteria will be explored, which may show the effectivity of a given heuristic. And finally, the measurements that were made will be discussed and evaluated according to the given criteria.

4.1 Benchmarks and benchmarking tool

One of possible sources of benchmarks is SMT-LIB [3], which collects a diverse set of benchmarks from different sources, size of input formula and number of benchmarks in each collection. The benchmarks used are available at [17]. But, used benchmarks are part of the artifact attached to this work.

The second drawback of the SMT-LIB benchmark sets is that no description of each set is provided and where the benchmarks were obtained. Some of the benchmarks have a source in the description of the file, but not all of them. So, the description of each set contains as much information as possible.

The important property of all benchmarks is that they are written in the SMT-LIBv2 language. There are two categories of benchmarks based on the theory used. The QF_S benchmarks incorporate only theory of string, whereas QF_SLIA are benchmarks that use both theory of strings with theory of linear arithmetic combined. The prefix QF stands for quantifier-free. All information about the benchmarks is obtained from the source files. When a source with a further description is available, it is also provided.

- *sygus* (QF_S) is set of benchmarks generated by CVC4 for testing a string solvers.

- *slog* (QF_S) is set generated by the Stranger generator and provides examples of string manipulation in a web application and was generated from real applications. It uses operations union, concatenation, and replacement [24].
- *Norn* (QF_SLIA) is set generated by Eldarica. In benchmark files it is described as „CEGAR based model checking for string programs“.
- *PyEx* (QF_SLIA) is sets of benchmark generated by PyEx, converted to the SMT-LIB format by CVC4 and provides a symbolic execution of Python programs.
- *Kaluza* (QF_SLIA) is generated by Kudzu and it provides benchmarks that are based on real JavaScript examples [21]. It is internally divided into small and big subsets, where the size in the name refers to the size of the formula. Also, it is divided into subsets sat and unsat, but the unsat subset contains both sat and unsat formulas.
- *Kepler* (QF_SLIA), generated by Quang Loc Le and contains examples of word equations based on a handcrafted examples [12].
- *full-str-int* (QF_SLIA) is generated by Pyconbyte and is based on Python examples. The used subset is the one for Z3-Trau.
- *slent* (QF_SLIA) is set of modified slog benchmarks that aims to analyze security of string manipulations.
- *Leetcode* (QF_SLIA) is generated with PyExZ3 and aims at a concolic execution of Python programs.

The tool used to run these benchmarks is smt-bench [23] which is based on the python script `pycobench`. It uses an input file, where the paths to all files of the given benchmark are provided, simple shell script, which wraps the solver, and a yaml file, that can be used to run more solvers in one run. In addition, a timeout can be set and a number of processes can be set to enable it to run in parallel.

The output of this script is the log file, which contains information about the start and end of the actual benchmark. A second script, `pyco-proc`, for evaluation is provided, which creates a summary of the run in the *comma separated value* (csv) format.

The drawback of the smt-bench tool is that it provides the time in only two decimal places. Due to that, some of the benchmarks were done in 0.00 seconds. Also, when the benchmark ends with timeout, it does not provide the time, but returns TO instead. These values have to be converted into numbers during post-processing.

4.2 Evaluation criteria

Based on the information provided in the articles about CVC4 and CVC5 assume the soundness of the solvers [13, 19] – *refutation soundness* (when solver returns `unsat`, formula is indeed unsatisfiable) and *model soundness* (returned model is actual T_S-consistent model of the formula). Thus, when the measurement is done, these statements are valid. The possible return values of the solvers are `sat, unsat, to` (`to` stands for timeout) and `unknown` (i.e., when the computation cannot infer new fact about the formula, but there is not enough information to decide satisfiability). To preserve these statistics, it shall be enough to avoid the transition of the result between `sat` and `unsat`. Therefore, the output of the modified solver is checked against the full solver to ensure that the provided results are valid. The

	sat	unsat	to	unknown
sat	✓	✗	✓	✓
unsat	✗	✓	✓	✓
to	✓	✓	✓	✓
unknown	✓	✓	✓	✓

Table 4.1: Comparative table for possible change in results of the solvers.

check is done by Python script that is driven by the Table 4.1. If there is a transition from column value to row value (or vice versa) the result is then marked as ✓, when it is permitted change, or marked as ✗, when it is a prohibited change.

Note that the table allows for the transition from `to` or `unknown` to (`un`)`sat`. Assume that the heuristics and rules are applied in a given order. Thus, it is possible that some of the computation of the heuristic takes too much time to successfully solve a problem in a given time. When this heuristic is disabled, the right rule is applied earlier, and thus problem that initaly end with timeout may become solvable.

This work aims to primarily measure the time complexity of the solver and how it is affected by given heuristics. So, the main criteria will be the overall time needed to compute the given set of benchmarks. Due to that, the benchmarks may aim for different parts of the solver, and there is a need to address that because the heuristic may affect just one set of benchmarks dramatically and does not affect another.

The modification performed on the solvers disables one or more of the heuristics or optimizations of the solver. Therefore, it is expected that most of the measurements will provide worse results than the full solver.

Based on the provided observation, a set of criteria is defined.

1. There is no prohibited result change.
2. The time consumption of the full solver is at least one of the below:
   (a) 2 % better in more than half of the sets of the benchmarks (general heuristic).
   (b) 5 % better in one of the sets of benchmarks (specific heuristic).

These criteria serve to find an effective heuristic. Thus, if there is a heuristic that satisfies at least one of the given time consumption criterion, it can be marked as an effective heuristic. The measurement aims at finding a set of effective heuristics.

Also, the solver runs in parallel, when it is possible, and given timeout to solve one benchmark is 20 seconds. The resulting time used in the tables provided is obtained by the sum of the run-times of all benchmarks. The disadvantage of summing all the times is that for some benchmarks the resulting run time is 0 due to the processing reasons mentioned earlier. This does create a cumulative error in the resulting run-time.

4.3 Rewriter heuristics measurements

The first measurements were performed using the blind approach. Each of the solvers has a complex rewriter, which simplifies the input formula. During these measurement process, it was monitored if any part of these rewriting rules have a high impact on the resulting time. The heuristics were disabled by commenting given part of the code and building a new binary. Typically, when the part of the code operated on constants, all these processing of

	CVC5					
	full	range_re2	concat2	eq_ext3	replace1	replace2
full_str_int	2476.21	2499.32	2509.91	2456.19	2454.91	2462.09
kaluza_big_sat	113.65	120.12	121.10	119.43	113.62	119.02
kaluza_big_unsat	811.22	849.43	840.66	821.90	832.04	822.95
kaluza_small_sat	3.32	3.41	3.26	3.37	3.35	3.31
kaluza_small_unsat	7.72	7.05	6.66	6.82	7.03	6.64
leetcode	128.66	131.27	134.88	129.52	131.85	131.84
norn	2044.74	2043.10	2034.55	2039.83	2036.65	2042.37
pyex_httplib2	1102.19	1131.59	1148.23	1131.48	1136.03	1129.34
pyex_httplib3	2558.64	2590.08	2644.25	2594.64	2592.38	2586.30
pyex_mongo3	874.05	885.27	908.30	880.71	883.12	881.38
pyex_pip	1.67	1.83	1.81	1.66	1.70	1.88
pyex_pymongo	83.57	84.23	84.33	83.61	86.22	83.29
pyex_pymongo2	247.02	254.43	255.69	251.35	252.63	252.78
pyex_requests3	11.24	11.93	12.31	11.05	11.70	11.99
pyex_url_request	13.38	13.71	13.38	13.20	13.81	13.73
slent	1295.17	1316.90	1308.79	1298.11	1303.05	1297.25
slog	9.84	10.05	9.96	9.56	9.69	9.59
sygus	132.45	148.31	141.43	143.45	140.78	142.62

Table 4.2: Time consumption CVC5 rewriter heuristics in seconds, that meet one of the conditions of effective heuristics. At least 2 % better is light green, at least 5 % better is darker green.

constants were disabled together. Unfortunately, the disablement was done in situ, without documenting which binary belongs to which heuristics. An attempt was made to reconstruct these parts in the CVC5 solver, but in the course of reconstruction more heuristics were found than in the original set. The rewriter file with the comments is provided in the attached digital artifact. Each part of the code that can be disabled is marked with comment `*turnoff <name>`, where the *name* is the name used in the results and in this work.

During these measurements, 86 binaries were measured for CVC5 and 163 binaries for Z3. The tests run on hardware with an Intel i7 4770 3.40GHz processor and 32GB of memory. The binaries were run on all provided benchmarks and the measurement took about 3 weeks. Because of the enormity of the measurement, only part of the results is provided here.

4.3.1 Results

In the first column of provided tables is always the full configuration of the given solver to be referenced. The heuristics that satisfy the general heuristic criterion (2a) in full configuration are emphasized in italics, and the heuristics that satisfy the specific heuristic criterion (2b) are highlighted in bold.

For the CVC5 solver, the first two heuristics (range_re2, concat2) in the Table 4.2 satisfied the general heuristic criterion. Thus, it has an impact on most of the computed benchmarks. The heuristics concat2, eq_ext3 and both replace satisfy the specific heuristic criterion.

	Z3					
	full	cont10	cont2	eqCore3	regexp3	substr8
full_str_int	2265.35	2345.59	2337.76	2214.43	2319.02	2316.09
kaluza_big_sat	29.66	32.27	33.72	41.19	90.69	32.96
kaluza_big_unsat	4374.07	4384.05	4386.00	4825.70	7042.46	4372.51
kaluza_small_sat	167.94	174.65	175.74	174.19	175.82	176.02
kaluza_small_uns	56.79	58.66	59.43	59.43	60.08	58.84
leetcode	85.08	90.44	89.90	83.37	87.87	87.85
norn	2591.88	2593.09	2592.50	2595.28	2591.55	2657.26
pyex_httplib2	3891.66	5409.37	4095.14	4040.63	4038.64	3974.41
pyex_httplib3	11 037.87	17 590.68	11 467.18	11 402.17	11 239.43	11 230.42
pyex_pip	0.96	1.04	1.00	1.06	1.02	1.00
pyex_pymongo2	1539.69	1573.93	1545.22	1495.50	1541.89	1540.86
pyex_url_request	241.97	280.97	256.32	247.10	249.20	247.22
slent	1749.61	1741.64	1755.28	1827.02	1982.39	1752.20
slog	925.10	929.21	929.57	968.58	937.94	922.96
sygus	16.57	17.44	17.55	17.43	17.27	17.40

Table 4.3: Time consumption of Z3 rewriter heuristics in seconds, that meet one of the condition of effective heuristics. At least 2 % better is light green, at least 5 % better is darker green.

In the attached digital artifacts is provided a list with all heuristics that satisfy the 5 % limit. It happened 126 times, and 44 times for the benchmark sygus. In addition, the heuristics range_re2 and concat2 are the only two that satisfied the (2a) criterion.

Note that for some of the disabled heuristics, the overall solving time is better than with them. But it is usually only in not many cases and not for the majority of tested benchmarks. Even the first two heuristics, that are effective in a significant number of benchmarks, have some negative impact on solving the kaluza_small_unsat.

The Table 4.3 provides an overview of the measurement of the heuristics of Z3. All provided heuristics are also all heuristics that satisfied the (2a) criterion. The (2b) criterion was satisfied 331 times, of which 112 were reached solving the kaluza_big_sat benchmark.

Based on the values provided in the table, it can be seen that some of the rewriter heuristics had a really high impact on the solving time. For example, the cont10 heuristic in the benchmark pyex_httplib3 has a high impact on effective solving. Also, for the same benchmark, the cont2 heuristic provides a better result at least 2 % . However, it may be caused by the frequent occurrence of the `contains` function in the benchmark.

Table 4.4 provides a comparison of the full configurations of both solvers. In general, CVC5is faster on most benchmarks. Note that for a number of benchmarks, CVC5 has a dramatically better solving time. However, there are 3 benchmarks where the Z3 performs significantly better. In the number of solved benchmarks, the solvers are not much different and in most cases provide a comparable number of successfully solved inputs. Note that for the pyex_httplib3 the runtime of Z3 is 4 times larger, but only 174 fewer `sat`s were obtained.

The worst result Z3 provided was on benchmark pyex_mongo3, where the time is more than 27 times worse than in CVC5 and also it resolves only about a third of the satisfiable formulas.

	CVC5_full				Z3_full			
	time[s]	sat	unsat	to	time[s]	sat	unsat	to
full_str_int	2476.21	558	461	30	**2265.35**	540	464	45
kaluza_big_sat	113.65	506	0	1	**29.66**	507	0	0
kaluza_big_unsat	**811.22**	1415	2319	11	4374.07	1244	2313	188
kaluza_small_sat	**3.32**	11382	0	0	167.94	11382	0	0
kaluza_small_u	**7.72**	2988	810	0	56.79	2988	810	0
leetcode	128.66	867	1785	0	**85.08**	866	1785	1
norn	**2044.74**	706	228	93	2591.88	712	188	127
pyex_httplib2	**1102.19**	3488	793	0	3891.66	3454	773	54
pyex_httplib3	**2558.64**	5079	631	18	11 037.87	4905	610	213
pyex_mongo3	**874.05**	1537	564	5	23 625.68	585	532	989
pyex_pip	1.67	31	3	0	**0.96**	31	3	0
pyex_pymongo	**83.57**	259	44	2	1007.5	217	44	44
pyex_pymongo2	**247.02**	259	256	8	1539.69	204	255	64
pyex_requests3	**11.24**	61	58	0	297.28	59	48	12
pyex_url_request	**13.38**	42	40	0	241.97	40	35	7
slent	**1295.17**	601	482	45	1749.61	565	481	82
slog	**9.84**	808	1168	0	925.1	771	1168	37
sygus	132.45	343	0	0	**16.57**	343	0	0

Table 4.4: Comparison of runtime and solving results between CVC5 and Z3. The faster solver is highlighted in bold.

4.4 Advanced heuristics measurements

The structure of the second further measurement is based on the results of the first. First of all, the rewriter heuristics does not have a significant impact on overall performance of the solver in isolation; only a few of them had a greater impact on the result, and for some cases the speedup provided by the specific heuristic is more significant.

Based on a comparison of the solvers, for each solver, only four sets of heuristics were chosen, in which the best result was given in comparison with the second solver. The criterion used was the speedup in time (e.g. CVC5 solved the slog more than 94 times faster than Z3).

The idea behind the second measurement was to find which of the heuristics provides the effectiveness of the solving the given set of benchmarks. Unfortunately, this approach was shown to be difficult to accomplish due to the structure of the tools.

The CVC5 provides a set of heuristics and approaches in the articles [19, 18, 20], where they compare the effectivity of a given heuristic with a previous iteration of the solver that does not have implemented that heuristic. Unfortunately, there is no possibility of disabling many of the heuristics in the newer iterations of the tools. The heuristics are usually embedded into the solver in a way, when other functions are dependent on the result. Many of the heuristics are described in the comments, but, for example, the witness sharing in CVC5 is not mentioned in the code. So it is possible to find implementation of a given heuristic but is is hard to disable it.

The Z3 is barely documented, so the purpose of the code has to be inferred from the comment, if it is available, name of the method, or from the code itself. The described approach of RE derivatives can be found in the rewriter of Z3, but the entry point returns

CVC5

	full				arith				extf			
	time[s]	sat	u	t	time[s]	sat	u	t	time[s]	sat	u	t
1	6.08	11382	0	0	5.17	11382	0	0	5.54	11382	0	0
2	725.49	1536	564	6	727.84	1536	564	6	727.89	1536	564	6
3	8.92	61	58	0	8.85	61	58	0	8.73	61	58	0
4	8.44	808	1168	0	8.12	808	1168	0	8.11	808	1168	0
5	2500.43	558	460	31	2513.20	558	460	31	2518.39	558	460	31
6	2415.66	5078	631	19	2424.00	5078	631	19	2428.57	5078	631	19

	fmf				rewriter			
	time[s]	sat	u	t	time[s]	sat	u	t
1	5.29	11382	0	0	4.98	11382	0	0
2	727.40	1536	564	6	727.83	1536	564	6
3	8.76	61	58	0	8.39	61	58	0
4	7.63	808	1168	0	7.56	808	1168	0
5	2512.76	558	460	31	2509.62	558	460	31
6	2426.52	5078	631	19	2432.55	5078	631	19

Table 4.5: Second measurement for CVC5. `u` means `unsat`, `t` means timeout. The benchmarks used are 1) kaluza_small_sat, 2) pyex_mongo3, 3) pyex_request3, 4) slog, 5) full_str_int, 6) pyex_httplib3.

only success on rewriting. Thus, disabling this part of the solver will probably lead to unsoudness of the solver. Also, many of the functions return either success and continue in the computation or fail, leading to the immediate end of solving and returning `unsat`.

For this measurement in CVC5 the inference arithmetic schema from Section 3.2.3, the reduction of the extended function and the finding of the upper bound of the quantifier from Section 3.2.2, and a complete rewriter was disabled.

In Z3 a pair of functions was found that is used to compute bounds on the string. Both of these functions were disabled. In addition, a complete rewriter was disabled. Other parts seem to be too interconnected to be disabled.

As in the previous section, all disabled heuristics are marked with the comment `*turnoff <name>` in the source files. The names are provided in tables.

For measurement was used a different machine with only 16GB of memory and with AMD Ryzen 5 5625U 2.30GHz, timeout was set to 20s.

4.4.1 Results

Because the measurement was performed on different hardware, the full configuration was also measured with the rest of the binaries. The time in the tables is provided in seconds, and a number of satisfiable inputs are provided to be able to compare everything together.

The results of CVC5 show that any of the disabled heuristics does not have an impact on the benchmarks 1-4. Because of that, the measurements were repeated with the same results. Therefore, the full_str_int and pyex_httplib3 benchmarks were added, as they both needed the most time to be solved. Both added benchmarks provide a result similar to the first four. Although the computation time of full CVC5 is about two and a half thousand seconds, the modified solvers were less than 20 seconds slower, so the resulting slowdown is less than 1 %.

For benchmarks 1, 3, and 4, the CVC5 runs slightly faster. Note that for benchmark 2, all modified binaries run for almost the same time. In addition, the number of satisfiable inputs remains unchanged for benchmarks 1-4.

	Z3											
	full				arithBounds				rewriter			
	time[s]	`sat`	`u`	`t`	time[s]	`sat`	`u`	`t`	time[s]	`sat`	`u`	`t`
a	2808.37	524	464	61	10 004.59	117	449	390	11 642.65	295	328	426
b	35.90	507	0	0	376.68	444	0	4	363.34	497	0	10
c	118.70	866	1785	1	2671.09	722	1770	125	444.48	868	1774	10
d	23.46	343	0	0	22.50	60	0	0	18.09	343	0	0

Table 4.6: Second measurements for Z3. The solver is unsound for arithBounds and rewriter disabled. The benchmarks are a) full_str_int, b) kaluza_big_sat, c) leetcode, d) sygus

The complete Z3 solver performed slightly worse on the second hardware, as can be seen in Table 4.6. Both disabled heuristics were shown to be important for the overall computation. But by disabling the rewriter and the arithBounds, the solver becomes unsoud. Some of the `unsat` inputs becomes *sat*. In addition, the time increased dramatically for both modified binaries. Along with a lower time efficiency, the number of satisfiable inputs decreased significantly.

4.5 Evaluation

The string solvers, both CVC5 and Z3, provide good results in the number of solved formulas. However, the time consumption of CVC5 is better in most cases.

The CVC5 seems to be a more stable solver. For some of the benchmarks, it is probably possible to solve the input formula efficiently and independently on the other parts of the solver. For the second measurement, it is somewhat contraintuitive that disabling such a large part of the solver leads to an increase in time efficiency. Some suitable heuristic, which decides which parts are needed to be used for solving a given input, may dramatically increase the efficiency of the solver. It is also possible that preprocessing of the input simplifies the formula enough to result in a successful solution.

The second measurement probably contains an error, because of the low impact of modifying the solver. But the time improvement in case of benchmarks with small input formula may show that the solver is also highly dependent on the other parts. Thus, a good integration and effectiveness of the other parts of the solver may play a crucial role in overall effectiveness, not only the heuristics implemented in the solver itself.

In addition, it seems that the CVC5 string solver is able to cooperate better as a whole. Even if some part of the procedure is missing, the solver is still able to successfully solve the input formula.

In comparison, the Z3 solver seems to be more monolithic, where each part of the whole is important for the process. The simplifying performed by rewriter seems to be essential for successful solving. Implements a bunch of heuristics that contribute to overall effectivity.

Unfortunately, the solver has a large undocumented codebase, where it is challenging to infer a purpose of each part.

In both of the solvers, it is unclear which part is essential for the computation. Although both of the solvers are DPLL(T) based, thus a lazy solver, they use many of the techniques

used in eager solving. It is clear that many of the implemented techniques are implemented by both of the solvers.

Unfortunately, based on research and measurement, it is almost impossible to draw a conclusion about the heuristics provided. The first measurements show that there is no significant part that provides a dramatic increase in speedup of solving. It is possible that choosing a set of simplifying rules is highly dependent on the algorithm that solves the constraints, thus not all rewriting rules have to be implemented in a particular solver. In addition, many of the rewrites depend on another inference schema, which guides the simplification.

In general, this work provides many approaches that can be used to implement the effective string solver. As a good combination for basic and extended string functions, the combination of simplifying the extended functions with reasoning about length bounds seems effective. One of the approaches that may provide good results with the lazy solver is to bound the possible alphabet to some smaller subset, which is described in [14]. The approximation of the string length used in CVC5 (Section 3.2.3) seems to provide good results along with the fact that integer arithmetic theory is also a complex problem. The approach introduced in [22] for the regular membership constraints appears to be one of the most effective approaches currently available. Also, a right selection of the base solver may also lead to significantly better results without having to optimize the string solving algorithm.

In conclusion, the measurements performed did not provide much information. One of the sources of uncertainty is the used benchmarking tool that provides time with only two decimal precision. This may lead to misinterpretation of the measured data as many of the benchmarks were solved literally in no time – the tool returned 0.0 as the consumed time in many cases. Increasing the accuracy of the time will make the measurements more precise, leading to a more accurate evaluation of the impact each heuristic has. And it is unclear when the tool rounds to 0, if it is when the value is smaller than 0.005, or if it rounds to 0 even when the number is 0.009. Also, this error leads to the given criteria should be specified better.

A possible enhancement is to make the criteria more precise, for example, set the limit for the general heuristic to at least 5 % better in more than half of the tested benchmarks. Then the result of the measurement is that there is no general heuristic in any of the rewriters. Therefore, to obtain some more specific information, the heuristics in rewriter should be divided into sets that are disabled at once. This can be the goal of future work.

Chapter 5

Conclusion

This work describes the basic principles of STM solving, providing background information on the theories and algorithms used. Introduces two SMT solvers – CVC5and Z3, which provide reasonably effective string solvers. Then, this work provides the principles of string solving and a set of of heuristics that are used in the described solvers. In the end, this thesis describes a set of measurements that were performed and discusses the results. In addition, some thoughts on the heuristics and approaches that can be used to create an effective solver are discussed at the end of the measurement chapter.

One of the goals of this thesis was to measure and evaluate the impact of heuristics and its combinations on sets of benchmarks. This goal was only partially met. The first of the measurements aimed to measure the impact of simplifying rules used in the rewriters of given solvers. The result of this measurement is that each of the heuristics in isolation does not have a great impact on the set of benchmarks used.

The second of the measurements aimed to find some different heuristics that may have a greater impact on the set. Due to the complexity of the tools, 4 heuristics in CVC5and only two heuristics in Z3 were identified. The measurement then shows that in CVC5 the heuristics does not have an impact on the initial chosen set. So, more benchmarks were added to the set and measurement was repeated, which did not change the result at all. The Z3 without the 2 heuristics became unsound as it marks some of the benchmarks as satisfiable although the complete tool marked them unsatisfiable. The unfulfilled part of the goal is to measure a combination of heuristics. In case of Z3, there was not identified a suitable set of heuristics. The combination of heuristics in CVC5 was not measured due to time constraints.

In case of the rewriter, it was concluded that better results may be obtained by dividing the heuristic into a set or classes and then disabling the whole set at once and repeating the measurements. This was not done because of the aim of the work was moved to identify a more complex heuristics. So, measuring the effectiveness of this part of the solvers can be a goal for future work.

The second goal was to determine which of these heuristics could be potentially used in the VeriFIT string solver. This goal depends on the success of the measurement that did not provide satisfactory results. However, potentially usable heuristics were discussed in the evaluation part of the measurements. The effectiveness of the heuristics cannot be justified due to the lack of data which reliably confirm its benefits.

In the following work, measurement of the combination of CVC5 heuristics from the second part of experiments can be done on all heuristic sets, as this part is time-consuming

only due to the runtime of all the benchmarks. Further future work can be done in the field of code analysis of the solvers to find more used heuristics.

Bibliography

[1] Aniva, L.; Barbosa, H.; Barrett, C.; Brain, M.; Camillo, V. et al. *Cvc5, version 1.0.5, commit `7d0db1a`.* Software. 2023. Available at: https://github.com/cvc5/cvc5. [cit. 2024-05-09].

[2] Barbosa, H.; Barrett, C.; Brain, M.; Kremer, G.; Lachnitt, H. et al. Cvc5: A Versatile and Industrial-Strength SMT Solver. In: Fisman, D. and Rosu, G., ed. *Tools and Algorithms for the Construction and Analysis of Systems.* Cham: Springer International Publishing, 2022, p. 415–442. ISBN 978-3-030-99524-9.

[3] Barret, C.; Fontaine, P. and Tinelli, C. *The Satisfiability Modulo Theories Library (SMT-LIB).* Online. 2016. Available at: https://smt-lib.org/. [cit. 2024-5-3].

[4] Barret, C.; Fontaine, P. and Tinelli, C. *The SMT-LIB Standard, Version 2.6.* Online. 2021. Available at: https://smt-lib.org/papers/smt-lib-reference-v2.6-r2021-05-12.pdf. [cit. 2024-05-03].

[5] Barret, C.; Sebastiani, R.; Seshia, S. A. and Tinelli, C. Satisfiability Modulo Theories. In: Bierre, A. et al., ed. *Handbook of satisfiability.* Amsterdam: IOS Press, 2009. Frontiers in artificial intelligence and applications; vol. 185. ISBN 978-1-58603-929-5.

[6] Barrett, C. and Tinelli, C. Satisfiability Modulo Theories. In: Clarke, E. M.; Henzinger, T. A.; Veith, H. and Bloem, R., ed. *Handbook of Model Checking.* Cham: Springer International Publishing, 2018, p. 305–343. ISBN 978-3-319-10575-8. Available at: https://doi.org/10.1007/978-3-319-10575-8_11.

[7] Bjørner, N.; Eishenhofer, C.; Gurfinkel, A.; Lopes, N. P.; de Moura, L. et al. *Z3 Internals (Draft).* Online. 2023. Available at: https://z3prover.github.io/papers/z3internals.html. [cit. 2024-04-29].

[8] Bjørner, N. and Nachmanson, L. Navigating the Universe of Z3 Theory Solvers. In: Carvalho, G. and Stolz, V., ed. *Formal Methods: Foundations and Applications.* Cham: Springer International Publishing, 2020, p. 8–24. ISBN 978-3-030-63882-5.

[9] Bjorner, N.; Wintersteiger, C. M. and Grigorenko, P. *Publications (Z3 GitHub).* Online. 2019. Available at: https://github.com/Z3Prover/z3/wiki/Publications. [cit. 2024-05-11].

[10] Bjorner, N.; Wintersteiger, C. M.; Nachmanson, L.; de Moura, L.; Berzish, M. et al. *Z3, version 4.12.2.0, commit `8a3a3dc`.* Online. 2023. Available at: https://github.com/z3prover/z3. [cit. 2024-05-11].

[11] de Moura, L. and Bjørner, N. Z3: An Efficient SMT Solver. In: Ramakrishnan, C. R. and Rehof, J., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, p. 337–340. ISBN 978-3-540-78800-3.

[12] Le, Q. L. and He, M. A Decision Procedure for String Logic with Quadratic Equations, Regular Expressions and Length Constraints. In: Ryu, S., ed. *Programming Languages and Systems*. Cham: Springer International Publishing, 2018, p. 350–372. ISBN 978-3-030-02768-1.

[13] Liang, T.; Reynolds, A.; Tinelli, C.; Barrett, C. and Deters, M. A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions. In: Biere, A. and Bloem, R., ed. *Computer Aided Verification*. Cham: Springer International Publishing, 2014, p. 646–662. ISBN 978-3-319-08867-9.

[14] Lotz, K.; Goel, A.; Dutertre, B.; Kiesl Reiter, B.; Kong, S. et al. Solving String Constraints Using SAT. In: Enea, C. and Lal, A., ed. *Computer Aided Verification*. Cham: Springer Nature Switzerland, 2023, p. 187–208. ISBN 78-3-031-37703-7.

[15] Monniaux, D. A Survey of Satisfiability Modulo Theory. In: Gerdt, V. P.; Koepf, W.; Seiler, W. M. and Vorozhtsov, E. V., ed. *Computer Algebra in Scientific Computing*. Cham: Springer International Publishing, 2016, p. 401–425. ISBN 978-3-319-45641-6.

[16] Nieuwenhuis, R.; Oliveras, A. and Tinelli, C. Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T). *J. ACM*. 1st ed. New York, NY, USA: Association for Computing Machinery, nov 2006, vol. 53, no. 6, p. 937–977. ISSN 0004-5411. Available at: https://doi.org/10.1145/1217856.1217859.

[17] Preiner, M.; Schurr, H.-J.; Barrett, C.; Fontaine, P.; Niemetz, A. et al. *SMT-LIB release 2023 (non-incremental benchmarks)*. Zenodo, february 2024. Available at: https://doi.org/10.5281/zenodo.10607722.

[18] Reynolds, A.; Nötzli, A.; Barrett, C. and Tinelli, C. High-Level Abstractions for Simplifying Extended String Constraints in SMT. In: Dillig, I. and Tasiran, S., ed. *Computer Aided Verification*. Cham: Springer International Publishing, 2019, p. 23–42. ISBN 978-3-030-25543-5.

[19] Reynolds, A.; Notzlit, A.; Barrett, C. and Tinelli, C. Reductions for Strings and Regular Expressions Revisited. In: Ivrii, A. and Strichman, O., ed. *2020 Formal Methods in Computer Aided Design (FMCAD)*. TU Wien Academic Press, 2020, p. 225–235. ISBN 978-3-85448-042-6.

[20] Reynolds, A.; Woo, M.; Barrett, C.; Brumley, D.; Liang, T. et al. Scaling Up DPLL(T) String Solvers Using Context-Dependent Simplification. In: Majumdar, R. and Kunčak, V., ed. *Computer Aided Verification*. Cham: Springer International Publishing, 2017, p. 453–474. ISBN 978-3-319-63390-9.

[21] Saxena, P.; Akhawe, D.; Hanna, S.; Mao, F.; McCamant, S. et al. A Symbolic Execution Framework for JavaScript. In: *2010 IEEE Symposium on Security and Privacy*. 2010, p. 513–528. ISBN 978-1-4244-6895-9.

[22] Stanford, C.; Veanes, M. and Bjørner, N. Symbolic Boolean derivatives for efficiently solving extended regular expression constraints. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2021, p. 620–635. PLDI 2021. ISBN 9781450383912. Available at: https://doi.org/10.1145/3453483.3454066.

[23] Síč, J.; Lengál, O.; Havlena, V. and Blahoudek, F. *smt-bench, commit 2b427355.* Online. 2023. Available at: https://github.com/VeriFIT/smt-bench.

[24] Wang, H.-E.; Tsai, T.-L.; Lin, C.-H.; Yu, F. and Jiang, J.-H. R. String Analysis via Automata Manipulation with Logic Circuit Representation. In: Chaudhuri, S. and Farzan, A., ed. *Computer Aided Verification*. Cham: Springer International Publishing, 2016, p. 241–260. ISBN 978-3-319-41528-4.