BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

STRING-FLOAT CONVERSIONS IN STRING CONSTRAINT SOLVING

KONVERZE MEZI FLOATY A STRING VE STRING CONSTRAINT SOLVINGU

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **TOMÁŠ PAŘÍZEK**
AUTOR PRÁCE

SUPERVISOR **Mgr. JURAJ SÍČ**
VEDOUCÍ PRÁCE

BRNO 2025

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

164377

Institut:	Department of Intelligent Systems (DITS)
Student:	**Pařízek Tomáš**
Programme:	Information Technology
Title:	**String-Float Conversions in String Constraint Solving**
Category:	Algorithms and Data Structures
Academic year:	2024/25

Assignment:

1. Study the string constraint solving methods used in the string solver Z3-Noodler, especially the algorithms for working with conversions between integers and strings.
2. Suggest a generalization of these algorithms to conversions between integers and float decimals. It will not be possible to solve the problem in complete generality; find a practical solution (in the context of the examples provided by Honeywell Aerospace).
3. Implement the proposed methods and evaluate on a suitable benchmark provided by the advisor.

Literature:

- Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, Juraj Síč: Cooking String-Integer Conversions with Noodles. SAT 2024: 14:1-14:19

Requirements for the semestral defence:
1,2

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

Supervisor:	**Síč Juraj, Mgr.**
Consultant:	Fiedor Jan, Ing., Ph.D.
Head of Department:	Kočí Radek, Ing., Ph.D.
Beginning of work:	1.11.2024
Submission deadline:	14.5.2025
Approval date:	31.10.2024

Abstract

This bachelor's thesis deals with the design and implementation of a method for solving string constraints with string-float conversion. Solving logical formulae containing strings is a key area used in program verification, security analysis of software applications, including web and blockchain solutions, or in automatic test generation. However, current SMT-solvers mainly support only conversions between strings and integers.

Therefore, this work extends the stabilization approach of the Z3-Noodler solver to include conversions between real numbers and strings, thus generalizing algorithms originally designed only for integers. The newly implemented extension is integrated into the Z3-Noodler solver and uses iterative refinement of regular constraints represented by automata until a stable solution satisfying the given constraints is found. Experimental results show that this new functionality solves problems with real numbers efficiently and achieves performance comparable to existing methods for integer inputs.

Abstrakt

Tato bakalářská práce se zabývá návrhem a implementací metody pro řešení řetězcových omezení s konverzemi mezi řetězci a reálnými čísly. Řešení logických formulí obsahujících řetězce je klíčovou oblastí využívanou při verifikaci programů, analýze bezpečnosti softwarových aplikací, včetně webových a blockchainových řešení nebo při automatickém generování testů. Současné SMT-solvery ovšem převážně podporují pouze konverze mezi řetězci a celými čísly.

Tato práce proto rozšiřuje stabilizační přístup solveru Z3-Noodler o převody mezi reálnými čísly a řetězci, čímž zobecňuje algoritmy původně navržené pouze pro čísla celá. Nově implementované rozšíření je integrováno do solveru Z3-Noodler a využívá iterativní zpřesňování regulárních omezení reprezentovaných automaty, dokud není nalezeno stabilní řešení splňující daná omezení. Experimentální výsledky ukazují, že tato nová funkcionalita řeší úlohy s reálnými čísly efektivně a dosahuje výkonu srovnatelného s existujícími metodami pro celočíselné vstupy.

Keywords

string-float conversion, Z3-Noodler, SMT solver, stabilization procedure

Klíčová slova

převod mezi stringem a floatem, Z3-Noodler, SMT řešič, stabilizační procedura

Reference

PAŘÍZEK, Tomáš. *String-Float Conversions in String Constraint Solving.* Brno, 2025. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Mgr. Juraj Síč

String-Float Conversions in String Constraint Solving

Declaration

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

.....
Tomáš Pařízek
May 18, 2025

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Mgr. Juraj Síč, for his patient guidance, insightful feedback, and unwavering support throughout the preparation of this thesis. His expertise and encouragement were invaluable to the successful completion of my work.

Contents

Chapter 1

Introduction

String constraint solving has become a crucial aspect of formal verification and software analysis, particularly as modern software systems grow increasingly string-intensive. Many security vulnerabilities, most notably SQL injection and Cross-Site Scripting (XSS)[15] – stem from improper manipulation of string data in web applications, scripting environments (e.g., JavaScript, PHP, Python), and even smart contracts. Given the high stakes of securing these applications, the ability to accurately and efficiently solve string constraints, including those imposed by regular expressions, word equations, and length constraints, has never been more important.

The task of solving formulae with string constraints is typically delegated to specialized tools, often known as string solvers. These tools may stand on their own or be integrated into widely used SMT (Satisfiability Modulo Theories) solvers such as Z3 and CVC5 [12]. By treating strings as first-class entities subject to logical operations, string solvers allow developers and researchers to systematically reason about string-related properties in code, detect potential flaws, and apply corrective measures or formal proofs of correctness [18].

Among these tools is Z3-Noodler, a module designed to extend the Microsoft Z3 solver [13] with an efficient approach to solving string constraints based on what is often referred to as a „stabilization procedure“. This procedure refines the regular constraints of individual variables by representing possible solutions as finite automata, iteratively narrowing down the automata until reaching a stable state. A key feature of Z3-Noodler is its support for conversions between strings and integers, providing a broader range of applications such as verifying the correctness of numerical inputs.

Although previous research has focused mainly on integer-string conversions, practical applications, particularly those in industry, frequently require the ability to handle decimal numbers. Converting between strings and floating-point numbers becomes essential in contexts where data inputs or intermediate results contain decimal values, which is prevalent in automated testing and verification systems that handle complex numerical operations. However, this added capability poses challenges due to the intricacies of decimal representation, precision, and rounding.

The primary goal of this work is therefore to generalize existing algorithms in Z3-Noodler to include conversions between strings and decimal (float) numbers. This involves extending and adapting the stabilization procedure to account for decimal formats, addressing potential precision and rounding concerns, and implementing these enhancements within the Z3-Noodler tool. The proposed solutions were being systematically tested on changed public sets of benchmarks. The proposed sets of benchmarks from Honeywell Aerospace were not provided.

The structure of this thesis reflects the multifaceted nature of the problem. In Chapter 2, we outline key theoretical foundations, providing formal definitions of string constraints, regular expressions, and finite automata. Chapter 3 presents the current state of the art in string constraint solving, detailing the roles of tools such as Ostrich, Z3, or Z3-Noodler. Chapter 4 describes our core contributions, focusing on the technical and theoretical adaptations needed to handle decimal float conversions. Chapter 5 offers a detailed overview of the implementation of these adaptations in Z3-Noodler, addressing both algorithmic choices and practical engineering considerations. Finally, in Chapter 6, we discuss the experimental evaluation of the modified solver on a suite of benchmarks, analyze the results, and highlight avenues for further optimization and research. Chapter 7 concludes this work by summarizing the main contributions and proposing future directions for advancing string-to-float conversions and related techniques in automated verification.

Chapter 2

Preliminaries

In this chapter, we introduce the problems of string-constraint solving and theories of linear arithmetic within the broader context of satisfiability modulo theories (SMT) solving and first-order theories. Then we highlight their significance for reliable string-to-float conversions.

String solving refers to the process of taking a set of logical constraints on string variables and looking for specific assignments (i.e. specific strings) that satisfy all the constraints, or proving that such an assignment does not exist.

SMT extends classical Boolean satisfiability (SAT) by allowing atoms in a logical formula to belong to a particular theory (e.g. linear arithmetic, data structures, bit vectors, string theory). The SMT solver typically uses a SAT solver for the Boolean layer and modularly connects a specialized theory solver (T-solver) for the relevant theory. These components iteratively communicate with each other until they reach a conclusion about the existence or non-existence of the model.

In the case where the internal SAT solver uses the classical DPLL [13] algorithm (Davis-Putnam-Logemann-Loveland), we speak of the DPLL(T) algorithm.

2.1 Automata Theory

Finite automata provide a compact, algorithmically convenient representation of regular languages [9] and are therefore one of the ways to implement modern SMT string solvers. By associating each string variable with an automaton that recognizes all of its currently admissible values, the solver can encode constraints as standard automata operations. The remainder of this section recalls the formal definition of a finite automaton and the deterministic and nondeterministic variants used throughout the thesis.

2.1.1 Language Theory

Given the strong connection between formal languages (which this chapter will discuss), we must first define several important concepts from the theory of formal languages.

Definition 1 (Alphabet) *An* alphabet *is any finite, nonempty set of symbols. We typically denote an alphabet by* Σ.

Definition 2 (String) *Given an alphabet* Σ*, a string over* Σ *is a finite sequence of symbols*

$$s = a_1 a_2 \cdots a_n, \quad a_i \in \Sigma.$$

We write $|s| = n$ for its length.

Definition 3 (Concatenation of Strings) *If s and t are strings over Σ, their* concatenation *$s \cdot t$ is the string obtained by appending t to s:*

$$s \cdot t \;=\; a_1 a_2 \cdots a_{|s|}\, b_1 b_2 \cdots b_{|t|},$$

where $s = a_1 \cdots a_{|s|}$, $t = b_1 \cdots b_{|t|}$.

Definition 4 (Empty String) *The empty string, denoted by ε, is the unique string of length zero. It satisfies*

$$|\varepsilon| = 0 \quad \textit{and} \quad a = a\varepsilon = \varepsilon a,\ a \in \Sigma$$

Definition 5 (Power and Kleene Star) *Given an alphabet Σ, we define:*

$$\Sigma^0 \;=\; \{\varepsilon\}, \quad \Sigma^1 \;=\; \Sigma, \quad \Sigma^2 \;=\; \Sigma\Sigma,\ \ldots$$

and in general

$$\Sigma^n \;=\; \underbrace{\Sigma\Sigma\cdots\Sigma}_{n\ \textit{times}}.$$

The Kleene star *of Σ is*

$$\Sigma^* \;=\; \bigcup_{n \geq 0} \Sigma^n,$$

that is, the set of all finite strings (including the empty string ε) over Σ.

Definition 6 (Language) *A* language *over an alphabet Σ is any subset of Σ^*. That is,*

$$\mathcal{L} \subseteq \Sigma^*.$$

Definition 7 (Concatenation of Languages) *If $\mathcal{L}_1$ and $\mathcal{L}_2$ are languages over Σ, their concatenation $\mathcal{L}_1\mathcal{L}_2$ is*

$$\mathcal{L}_1\mathcal{L}_2 \;=\; \{\, s \cdot t \mid s \in \mathcal{L}_1,\ t \in \mathcal{L}_2 \} \;\subseteq\; \Sigma^*.$$

Definition 8 (Union of Languages) *Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be languages over an alphabet Σ. Their union is*

$$\mathcal{L}_1 \cup \mathcal{L}_2 \;=\; \{\, w \in \Sigma^* \mid w \in \mathcal{L}_1 \;\vee\; w \in \mathcal{L}_2 \}.$$

Definition 9 (Intersection of Languages) *Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be languages over an alphabet Σ. Their intersection is*

$$\mathcal{L}_1 \cap \mathcal{L}_2 \;=\; \{\, w \in \Sigma^* \mid w \in \mathcal{L}_1 \;\wedge\; w \in \mathcal{L}_2 \}.$$

Definition 10 (Complement of a Language) *Let L be a language over an alphabet Σ. The complement of $\mathcal{L}$ (with respect to Σ^*) is*

$$\overline{\mathcal{L}} \;=\; \Sigma^* \setminus \mathcal{L} \;=\; \{\, w \in \Sigma^* \mid w \notin \mathcal{L} \}.$$

2.1.2 Finite Automata

A finite automaton (FA) is a device that has a finite number of states and, when reading an input string, transitions between states according to a transition function [9].

Definition 11 (Nondeterministic Finite Automaton) *A nondeterministic finite automaton is a quintuple*

$$\mathcal{A} = \langle Q,\, \Sigma,\, \delta,\, q_0,\, F \rangle$$

where

- Q *is a finite set of states,*
- Σ *is the input alphabet,*
- $\delta : Q \times \Sigma \to 2^Q$ *is the transition function,*
- $q_0 \in Q$ *is the initial state,*
- $F \subseteq Q$ *is the set of accepting (final) states.*

We say $\mathcal{A}$ accepts *a string* $w \in \Sigma^*$ *if, when processing* w *from* q_0*, it ends in a state of* F*. The language of* $\mathcal{A}$ *is* $\mathcal{L}(\mathcal{A}) = \{\, w \mid \mathcal{A} \text{ accepts } w \}$.

Definition 12 (Deterministic Finite Automaton (DFA)) *A DFA is a finite automaton* $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F \rangle$ *that satisfies*

$$\forall\, q \in Q,\ a \in \Sigma : \quad \delta(q, a) \subseteq Q$$

In other words, for each (q, a) *there is exactly one defined transition.*

From classical automata theory, DFAs and NFAs recognize exactly the same class of languages (the regular languages) [19]. In practice, NFAs are often used directly in SMT string solvers, despite the existence of the powerset construction [17] to convert an NFA into an equivalent DFA, because they tend to be more compact for many practical problems.

2.1.3 Automata Operations

Now we define automata operations which are equivalent to language operations, and we use them for language operation representation.

Definition 13 (Intersection of NFAs) *Let* $\mathcal{A} = \langle Q_\mathcal{A}, \Sigma, \Delta_\mathcal{A}, q_\mathcal{A}, F_\mathcal{A} \rangle$ *and* $\mathcal{B} = \langle Q_\mathcal{B}, \Sigma, \Delta_\mathcal{B}, q_\mathcal{B}, F_\mathcal{B} \rangle$ *be NFAs with* $\Delta_\mathcal{A}, \Delta_\mathcal{B} \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$*. WE define the product NFA*

$$\mathcal{A} \cap \mathcal{B} = \big\langle Q_\mathcal{A} \times Q_\mathcal{B},\ \Sigma,\ \Delta,\ (q_\mathcal{A}, q_\mathcal{B}),\ F_\mathcal{A} \times F_\mathcal{B} \big\rangle,$$

where, for every $a \in \Sigma \cup \{\varepsilon\}$ *and* $(p, q) \in Q_\mathcal{A} \times Q_\mathcal{B}$*,*

$$\Delta\big((p, q), a\big) = \big\{ (p', q') \mid (p, a, p') \in \Delta_\mathcal{A} \wedge (q, a, q') \in \Delta_\mathcal{B} \big\}.$$

Language $\mathcal{L}(\mathcal{A} \cap \mathcal{B}) = \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$.

Definition 14 (Union of NFAs) *With* $\mathcal{A}$ *and* $\mathcal{B}$ *as above, we construct the NFA*

$$\mathcal{A} \cup \mathcal{B} = \big\langle Q_\mathcal{A} \cup Q_\mathcal{B} \cup \{q_{new}\},\ \Sigma,\ \Delta',\ q_{new},\ F_\mathcal{A} \cup F_\mathcal{B} \big\rangle,$$

where $\Delta' = \Delta_\mathcal{A} \cup \Delta_\mathcal{B} \cup \{(q_{new}, \varepsilon, q_\mathcal{A}),\ (q_{new}, \varepsilon, q_\mathcal{B})\}$*. Language* $\mathcal{L}(\mathcal{A} \cup \mathcal{B}) = \mathcal{L}(\mathcal{A}) \cup \mathcal{L}(\mathcal{B})$.

Definition 15 (Complement of an NFA) *Let $\mathcal{A} = \langle Q, \Sigma, \Delta, q_0, F \rangle$ be a non-deterministic finite automaton (NFA). An NFA for the complement language $\overline{\mathcal{L}(\mathcal{A})}$ can be obtained as follows:*

1. ***Determinisation and completion.***

   - *Apply the* subset construction *to turn $\mathcal{A}$ into an equivalent DFA*

     $$\mathsf{Det}(\mathcal{A}) \;=\; \big\langle\, 2^Q,\ \Sigma,\ \delta',\ \{q_0\},\ F' \big\rangle,$$

     where

     $$\delta'(S,a) \;=\; \bigcup_{q \in S} \{\, q' \mid (q,a,q') \in \Delta \},\quad F' = \{\, S \subseteq Q \mid S \cap F \neq \emptyset \}.$$

   - *To ensure the DFA is complete (every state has a defined transition for each $a \in \Sigma$), add a fresh sink state $\bot$. Redirect any undefined $\delta'(S,a)$ to $\bot$, and add self-loops $\delta'(\bot, a) = \bot$ for all $a \in \Sigma$.*

2. ***Swap accepting and non-accepting states.***

   $$\overline{\mathcal{A}} \;=\; \big\langle\, 2^Q \cup \{\bot\},\ \Sigma,\ \delta',\ \{q_0\},\ \big(2^Q \cup \{\bot\}\big) \setminus F' \big\rangle.$$

   In this machine, every state that was accepting in $\mathsf{Det}(\mathcal{A})$ becomes non-accepting, and every non-accepting state becomes accepting. The resulting DFA accepts exactly those strings over Σ that $\mathcal{A}$ does not accept.

Definition 16 (Concatenation of NFAs) *For NFAs $\mathcal{A}$ and $\mathcal{B}$ over Σ, we define*

$$\mathcal{A} \cdot \mathcal{B} = \big\langle Q_{\mathcal{A}} \cup Q_{\mathcal{B}},\ \Sigma,\ \Delta_{\mathcal{A}} \cup \Delta_{\mathcal{B}} \cup \{(p, \varepsilon, q_{\mathcal{B}}) \mid p \in F_{\mathcal{A}}\},\ q_{\mathcal{A}},\ F_{\mathcal{B}} \big\rangle.$$

Language $\mathcal{L}(\mathcal{A} \cdot \mathcal{B}) = \mathcal{L}(\mathcal{A}) \cdot \mathcal{L}(\mathcal{B})$.

2.1.4 Regular Expressions

Regular expressions provide an algebraic way to describe regular languages.

Definition 17 (Regular Expression) *Let Σ be a finite alphabet. The set of regular expressions over Σ is generated by the grammar [11]*

$$\mathcal{R} \;::=\; \emptyset \mid \varepsilon \mid a \mid \mathcal{R} + \mathcal{R} \mid \mathcal{R} \cdot \mathcal{R} \mid \mathcal{R}^*$$

where

- *$\emptyset$ denotes the empty language,*
- *ε denotes the language $\{\varepsilon\}$ containing only the empty string,*
- *$a \in \Sigma$ denotes the language $\{a\}$,*
- *$\mathcal{R}_1 + \mathcal{R}_2$ is union, $\mathcal{R}_1 \cdot \mathcal{R}_2$ is concatenation of languages,*
- *$\mathcal{R}^*$ is Kleene star (zero or more repetitions of strings from $\mathcal{L}(\mathcal{R})$).*

Definition 18 (Language of a Regular Expression) *Each regular expression R denotes a language $L(R) \subseteq \Sigma^*$ defined inductively:*

$$\begin{aligned}
\mathcal{L}(\emptyset) &= \emptyset, \\
\mathcal{L}(\varepsilon) &= \{\varepsilon\}, \\
\mathcal{L}(a) &= \{a\} \quad (a \in \Sigma), \\
\mathcal{L}(\mathcal{R}_1 + \mathcal{R}_2) &= \mathcal{L}(\mathcal{R}_1) \cup \mathcal{L}(\mathcal{R}_2), \\
\mathcal{L}(\mathcal{R}_1 \cdot \mathcal{R}_2) &= \mathcal{L}(\mathcal{R}_1) \cdot \mathcal{L}(\mathcal{R}_2) = \{\, uv \mid u \in \mathcal{L}(\mathcal{R}_1),\, v \in \mathcal{L}(\mathcal{R}_2)\}, \\
\mathcal{L}(\mathcal{R})^n &= \mathcal{L}(\underbrace{\mathcal{R} \cdot \ldots \cdot \mathcal{R}}_{n \text{ times}}) \\
\mathcal{L}(\mathcal{R}^*) &= \bigcup_{n \geq 0} \mathcal{L}(\mathcal{R})^n.
\end{aligned}$$

Equivalence to Automata. For every regular expression $\mathcal{R}$ there exists an NFA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{R}) = \mathcal{L}(\mathcal{A})$ [1], and conversely, every NFA (or DFA) admits an equivalent regular expression. Hence, regular expressions, DFAs, and NFAs all characterize the class of regular languages.

2.2 First Order Logic and Theories

In this section, we will introduce the concept of many-sorted first-order theories in general [10], and then we will also introduce specific theories used in this thesis.

Definition 19 (Many-Sorted First-Order Logic) *Many-sorted first-order logic generalizes classical first-order logic [5] by assigning each variable, function symbol, and predicate symbol a specific sort. Formally, a many-sorted signature is a triple*

$$\Gamma \;=\; (\mathcal{S}, \mathcal{F}, \mathcal{P}),$$

where

- *$\mathcal{S}$ is a non-empty (typically finite) set of sorts.*
- *$\mathcal{F}$ is a set of function symbols. Each $f \in \mathcal{F}$ has type $s_1 \times \cdots \times s_n \;\to\; s$ written $f_{s_1,\ldots,s_n \to s}$ where $s_1, \ldots, s_n \in \mathcal{S}$.*
- *$\mathcal{P}$ is a set of predicate symbols. Each $P \in \mathcal{P}$ has type $s_1 \times \cdots \times s_n$ written $P_{s_1,\ldots,s_n}$ where $s_i, \ldots, s_n \in \mathcal{S}$.*

Definition 20 (Variable Set) *For each sort $s \in \mathcal{S}$, let $\mathcal{V}_s$ be a countable set of variables of sort s, and let*

$$\mathcal{V} \;=\; \bigcup_{s \in \mathcal{S}} \mathcal{V}_s.$$

Definition 21 (Arity) *For each predicate symbol $P \in \mathcal{P}$ of type $s_1 \times \ldots \times s_n$ or function symbol $f \in \mathcal{F}$ of type $s_1 \times \ldots \times s_n \to s$, respectively, we define the arity as n and denote $P_{/n}$ or $f_{/n}$, respectively.*

Constants are simply 0-ary function symbols $c \in \mathcal{F}$ of type $\to s$.

Definition 22 (Term) *A term of sort s is defined by the following Backus–Naur Form:*

$$t_s \;::=\; x_s \;\mid\; c_s \;\mid\; f_{s_1,\ldots,s_n\to s}\big(t_{s_1},\ldots,t_{s_n}\big)$$

where

- $x_s \in \mathcal{V}_s$ *is a variable of sort* s,
- $c_s \in \mathcal{F}$ *is a constant of sort* s,
- $f_{s_1,\ldots,s_n\to s} \in \mathcal{F}$ *is an* n*-ary function symbol mapping sorts* $(s_1,\ldots,s_n)$ *to* s*, and each* t_{s_i} *is a term of sort* s_i.

Definition 23 (Many-Sorted First-Order Formula) *MSFO Formulae are then built from Boolean connectives and quantifiers in the following way:*

$$\varphi \;::=\; t_1 = t_2 \;\mid P_{s_1,\ldots,s_n}\big(t_{s_1},\ldots,t_{s_n}\big) \;\mid \neg\varphi \;\mid (\varphi\wedge\varphi) \;\mid (\varphi\vee\varphi) \;\mid (\varphi\to\varphi) \;\mid \forall x_s.\,\varphi \;\mid \exists x_s.\,\varphi$$

where $P_{s_1,\ldots,s_n} \in \mathcal{P}$ *is a predicate symbol of type* $s_1\times\cdots\times s_n$ *and* $x_s \in \mathcal{V}_s$.

Depending on whether quantifiers $\forall$ or $\exists$ occur in φ, we distinguish between quantified and quantifier-free formulae.

Definition 24 (Structure) *Given a signature* $\Gamma = (\mathcal{S},\mathcal{F},\mathcal{P})$*, a* Γ*-structure* $\mathcal{M}$ *consists of:*

- *For each sort* $s \in \mathcal{S}$*, a nonempty set* D_s*, called the domain of* s.
- *For each function symbol* $f_{s_1,\ldots,s_n\to s} \in \mathcal{F}$*, a concrete function*
$$f^{\mathcal{M}} :\; D_{s_1}\times\cdots\times D_{s_n} \;\to\; D_s.$$
- *For each predicate symbol* $P_{s_1,\ldots,s_n} \in \mathcal{P}$*, a relation*
$$P^{\mathcal{M}} \;\subseteq\; D_{s_1}\times\cdots\times D_{s_n}.$$

We denote the collection of all domains by $\mathsf{Dom}(\mathcal{M}) = \{D_s \mid s\in\mathcal{S}\}$.

Definition 25 (Variable Assignment) *Let* $\mathcal{M}$ *be a* Γ*-structure and let* $\mathcal{V}_s$ *be the set of variables of sort* s*. A variable assignment is a function*

$$\alpha \;:\; \bigcup_{s\in\mathcal{S}} \mathcal{V}_s \;\to\; \bigcup_{s\in\mathcal{S}} D_s$$

such that $\alpha(x) \in D_s$ *whenever* $x \in \mathcal{V}_s$.

Definition 26 (Satisfaction) *Given a* Σ*-structure* $\mathcal{M}$ *and a variable assignment* α*, the satisfaction relation* $\mathcal{M},\alpha \models \varphi$ *("*φ *holds in* $\mathcal{M}$ *under* α*") is defined inductively:*

$$\begin{aligned}
&\mathcal{M},\alpha \models t_1 = t_2 \quad \textit{iff} \quad t_1^{\mathcal{M}}(\alpha) = t_2^{\mathcal{M}}(\alpha),\\
&\mathcal{M},\alpha \models P(t_1,\ldots,t_n) \quad \textit{iff} \quad \big(t_1^{\mathcal{M}}(\alpha),\ldots,t_n^{\mathcal{M}}(\alpha)\big) \in P^{\mathcal{M}},\\
&\mathcal{M},\alpha \models \neg\varphi \quad \textit{iff} \quad \textit{not } \big(\mathcal{M},\alpha \models \varphi\big),\\
&\mathcal{M},\alpha \models \varphi_1\wedge\varphi_2 \quad \textit{iff} \quad \mathcal{M},\alpha \models \varphi_1 \textit{ and } \mathcal{M},\alpha \models \varphi_2,\\
&\mathcal{M},\alpha \models \exists x_s.\,\varphi \quad \textit{iff} \quad \exists d\in D_s :\; \mathcal{M},\alpha[x_s\mapsto d] \models \varphi,
\end{aligned}$$

and analogously for $\vee,\to,\forall$*. If* φ *has no free variables, we simply write* $\mathcal{M}\models\varphi$.

Notation. *For any term* t*, the expression* $t^{\mathcal{M}}(\alpha)$ *denotes the value obtained by:*

1. *evaluating each variable x in t as $\alpha(x) \in D_s$ where s is sort of x ($x \in \mathcal{V}_s$), and*
2. *applying the function interpretations $f^{\mathcal{M}}$ according to the syntax of t.*

In particular, if t is a constant symbol c then $t^{\mathcal{M}}(\alpha) = c^{\mathcal{M}}$ where $c^{\mathcal{M}}$ is some nullary function.

Definition 27 (Model) *A model of a closed formula φ is a Γ-structure $\mathcal{M}$ such that $\mathcal{M} \models \varphi$.*

Definition 28 (Satisfiability) *A closed formula φ is satisfiable if there exists a Γ-structure $\mathcal{M}$ with $\mathcal{M} \models \varphi$. Otherwise, φ is unsatisfiable.*

Definition 29 (Theory) *A theory $\mathcal{T}$ over Γ is a set of closed formulas. A structure $\mathcal{M}$ is a model of $\mathcal{T}$ (written $\mathcal{M} \models \mathcal{T}$) if $\mathcal{M} \models \psi$ for every $\psi \in \mathcal{T}$. $\mathcal{T}$ is satisfiable if it has at least one model. $\mathcal{T}$ entails a sentence φ (written $\mathcal{T} \models \varphi$) if every model of $\mathcal{T}$ is also a model of φ.*

2.2.1 Theories of Linear Arithmetic

Let $\Sigma = (\mathcal{S}, \mathcal{F}, \mathcal{P})$ be a many-sorted signature as in definition 19. We now isolate two important subsignatures and their corresponding theories.

Definition 30 (Signature of Linear Integer Arithmetic) *The signature*

$$\Sigma_{\text{LIA}} \;=\; (\mathcal{S}_{\text{LIA}}, \mathcal{F}_{\text{LIA}}, \mathcal{P}_{\text{LIA}})$$

is defined by

- $\mathcal{S}_{\text{LIA}} = \{\mathbb{Z}\}$.
- $\mathcal{F}_{\text{LIA}} = \{ +\colon \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z},\ -\colon \mathbb{Z} \to \mathbb{Z},\ \mu_k \colon \mathbb{Z} \to \mathbb{Z} \mid k \in \mathbb{Z} \setminus \{0\} \}$. *Here $\mu_k(x)$ denotes scalar multiplication by the integer k.*
- $\mathcal{P}_{\text{LIA}} = \{ =\colon \mathbb{Z} \times \mathbb{Z},\ <\colon \mathbb{Z} \times \mathbb{Z},\ \leq\colon \mathbb{Z} \times \mathbb{Z},\ >\colon \mathbb{Z} \times \mathbb{Z},\ \geq\colon \mathbb{Z} \times \mathbb{Z}$. *The predicate $\equiv_k (x, y)$ holds iff x and y are congruent modulo k.*

Definition 31 (Theory of Linear Integer Arithmetic) *The* theory of linear integer arithmetic*, denoted $\mathcal{T}_{\text{LIA}}$, is the set of all quantifier-free Σ_{LIA}-formulae that are true in the standard structure $\mathbb{Z}$ (with the usual interpretation of $0, +, -, \mu_k, =, <, \leq, \ldots$).*

Definition 32 (Signature of Linear Real Arithmetic) *The signature*

$$\Sigma_{\text{LRA}} \;=\; (\mathcal{S}_{\text{LRA}}, \mathcal{F}_{\text{LRA}}, \mathcal{P}_{\text{LRA}})$$

is defined by

- $\mathcal{S}_{\text{LRA}} = \{\mathbb{R}\}$.
- $\mathcal{F}_{\text{LRA}} = \{\, q :\to \mathbb{R} \mid q \in \mathbb{Q} \} \;\cup\; \{ +\colon \mathbb{R} \times \mathbb{R} \to \mathbb{R},\ -\colon \mathbb{R} \to \mathbb{R},\ \mu_r \colon \mathbb{R} \to \mathbb{R} \mid r \in \mathbb{Q} \}$. *Here q is a 0-ary symbol for the rational constant q, and $\mu_r(x) = r \cdot x$.*
- $\mathcal{P}_{\text{LRA}} = \{ =\colon \mathbb{R} \times \mathbb{R},\ <\colon \mathbb{R} \times \mathbb{R},\ \leq\colon \mathbb{R} \times \mathbb{R},\ >\colon \mathbb{R} \times \mathbb{R},\ \geq\colon \mathbb{R} \times \mathbb{R},\ \texttt{is\_int} : \mathbb{R}\}$.

Predicates from $=$ to $\geq$ have usual semantics, `is_int` holds iff the real number has its decimal part equal to 0.

Definition 33 (Theory of Linear Real Arithmetic) *The* theory of linear real arithmetic*, denoted $\mathcal{T}_{\text{LRA}}$, is the set of all quantifier-free Σ_{LRA}-formulae that are true in the standard structure $\mathbb{R}$ (or equivalently $\mathbb{Q}$ when restricted to rational coefficients).*

Remark. Both $\mathcal{T}_{\text{LIA}}$ and $\mathcal{T}_{\text{LRA}}$ are decidable [6]: $\mathcal{T}_{\text{LIA}}$ is essentially Presburger arithmetic [16] (complexity doubly-exponential in the worst case), and $\mathcal{T}_{\text{LRA}}$ admits polynomial-time decision procedures via the simplex algorithm [4] or Fourier-Motzkin elimination [7]. In SMT practice, these theories are central:

- $\mathcal{T}_{\text{LIA}}$ models discrete program quantities (loop counters, array indices, string lengths).
- $\mathcal{T}_{\text{LRA}}$ captures real-valued constraints (timing, physical measurements, floating-point error estimates).

2.2.2 Theory of Strings

Let Σ be a finite alphabet, and Σ^* the set of all finite strings over Σ. We formalize the theory of strings as follows.

Definition 34 (Signature of the Theory of Strings) *The many-sorted signature*

$$\Sigma_{\mathsf{Str}} \;=\; (\mathcal{S}_{\mathsf{Str}},\, \mathcal{F}_{\mathsf{Str}},\, \mathcal{P}_{\mathsf{Str}})$$

is given by

$$\begin{aligned}
\mathcal{S}_{\mathsf{Str}} &= \{\mathsf{Str}, \mathbb{Z}, \mathsf{REGEX}\},\\
\mathcal{F}_{\mathsf{Str}} &= \big\{\, \varepsilon :\to \mathsf{Str},\ \mathsf{concat} : \mathsf{Str}\times\mathsf{Str}\to\mathsf{Str},\ |\cdot| : \mathsf{Str}\to\mathbb{Z},\\
&\qquad\ \mathsf{substr} : \mathsf{Str}\times\mathbb{Z}\times\mathbb{Z}\to\mathsf{Str},\ \mathsf{indexOf} : \mathsf{Str}\times\mathsf{Str}\to\mathbb{Z},\\
&\qquad\ \mathsf{replace} : \mathsf{Str}\times\mathsf{Str}\times\mathsf{Str}\to\mathsf{Str}\big\},\\
\mathcal{P}_{\mathsf{Str}} &= \{=\!: \mathsf{Str}\times\mathsf{Str}\}\ \cup\ \big\{\in\!: \mathsf{Str}\times\mathsf{REGEX}\big\},
\end{aligned}$$

Semantics

- ε is the empty string,
- $\mathsf{concat}(x, y) = x \cdot y$ (associative, with ε as identity),
- $|x|$ returns the length of x (a bridge to $\mathcal{T}_{\text{LIA}}$ from definition 31),
- each $x \in \mathcal{R}$ tests membership of a string in a regular language represented by regular expression $\mathcal{R}$.

Definition 35 (Theory of Strings) *The theory of strings, $\mathcal{T}_{\mathsf{Str}}$, is the set of all quantifier-free Σ_{Str}-formulae that are true in the standard structure*

$$\big(\Sigma^*,\ \varepsilon,\ \mathsf{concat},\ |\cdot|,\ \mathsf{substr},\ \mathsf{indexOf},\ \mathsf{replace},\ \in_{\mathcal{L}}\big).$$

2.2.3 Combining Arithmetic Theories with String Theories

When an SMT problem involves both arithmetic constraints (e.g., over integers or reals) and string constraints, solvers must combine the theory of strings with the theory of linear arithmetic [2]. This is often done following the Nelson-Oppen framework [14] for the combination of decision procedures, provided the theories satisfy the convexity and other conditions required for the framework's smooth operation.

Length constraints linking to arithmetic: A crucial link between string and arithmetic theories is the $|\cdot|$ function, which maps string variables to integer variables representing

their length. For instance, a constraint $|x| + |y| = 10$ is interpreted in the linear integer arithmetic (LIA) domain, while x and y remain strings over Σ^*. Whenever the string solver reasons about x and y being concatenated or matched against patterns, it sends derived length constraints (like $|x \cdot y| = |x| + |y|$) to the arithmetic solver. Conversely, the arithmetic solver might deduce bounds on $|x|$) or prove certain numeric constraints unsatisfiable, which in turn prunes the search space for string assignments. In chapter 4 we will introduce additional function symbols that convert between the string/sequence theories and $\mathcal{T}_{\text{LRA}}$, making these theories interoperable in a combined SMT setting.

At first, we define the many-sorted first-order theory as follows:

Definition 36 (Combined Signature) *Let*

$$\Sigma_{\mathsf{Comb}} = (\mathcal{S}_{\mathsf{Comb}}, \mathcal{F}_{\mathsf{Comb}}, \mathcal{P}_{\mathsf{Comb}})$$

where

$$\begin{aligned} \mathcal{S}_{\mathsf{Comb}} &= \{\mathsf{Str},\ \mathbb{Z},\ \mathbb{R},\ \mathsf{REGEX}\}, \\ \mathcal{F}_{\mathsf{Comb}} &= \mathcal{F}_{\mathsf{Str}} \cup \mathcal{F}_{\text{LIA}} \cup \mathcal{F}_{\text{LRA}} \\ &\quad \cup \big\{\, \mathsf{to\_int} : \mathsf{Str} \to \mathbb{Z},\ \mathsf{from\_int} : \mathbb{Z} \to \mathsf{Str}, \\ &\qquad \mathsf{to\_real} : \mathsf{Str} \to \mathbb{R},\ \mathsf{from\_real} : \mathbb{R} \to \mathsf{Str}, \big\}. \\ \mathcal{P}_{\mathsf{Comb}} &= \mathcal{P}_{\mathsf{Str}} \cup \mathcal{P}_{\text{LIA}} \cup \mathcal{P}_{\text{LRA}}. \end{aligned}$$

Here $\mathcal{F}_{\mathsf{Str}}, \mathcal{P}_{\mathsf{Str}}$ are from definition 34, $\mathcal{F}_{\text{LIA}}, \mathcal{P}_{\text{LIA}}$ from definition 30, and $\mathcal{F}_{\text{LRA}}, \mathcal{P}_{\text{LRA}}$ from definition 32. The conversion function will be discussed in section 2.2.3.

Definition 37 (Combined Theory) *The combined theory $\mathcal{T}_{\mathsf{Comb}}$ is the set of all quantifier-free Σ_{Comb}-formulae true in the standard structure $\big(\Sigma^*, \mathbb{Z}, \mathbb{R}, \dots\big)$.*

Definition 38 (Assignment) *Let $\mathcal{V}_{\mathsf{Str}}$ be a finite set of string variables and Σ a finite alphabet. An* assignment *(or valuation) is a total function*

$$\nu : \mathcal{V}_{\mathsf{Str}} \to \Sigma^*$$

that maps every variable $x \in \mathcal{V}_{\mathsf{Str}}$ to a concrete string $\nu(x)$ over the alphabet Σ.

Definition 39 (Language Assignment) *Let $\mathcal{V}_{\mathsf{Str}} = \{\, w, x, y, z, \dots \}$ be a finite set of string variables. To each variable $x \in \mathcal{V}_{\mathsf{Str}}$, we assign a regular language*

$$\mathit{Lang}(x) \subseteq \Sigma^*,$$

which describes the set of admissible string values for x. This function

$$\mathit{Lang} : \mathcal{V}_{\mathsf{Str}} \to 2^{\Sigma^*}$$

serves as the symbolic domain definition for each variable, typically represented by a regular expression or NFA.

Definition 40 (Substitution Mapping) *Let $\sigma : \mathcal{V}_{\mathsf{Str}} \to \mathcal{V}_{\mathsf{Str}}^*$ be a substitution mapping each string variable to a finite sequence of other variables. That is, $\sigma(x) = x_1 x_2 \dots x_n$, where $x_i \in \mathcal{V}_{\mathsf{Str}}$, and $\mathcal{V}_{\mathsf{Str}}^*$ is the free monoid over $\mathcal{V}_{\mathsf{Str}}$.*

We interpret σ with respect to a concrete assignment $\nu : \mathcal{V}_{\mathsf{Str}} \to \Sigma^*$ such that:

$$\nu(x) = \nu(x_1) \cdot \nu(x_2) \cdot \dots \cdot \nu(x_n),$$

where $\cdot$ denotes string concatenation. If $\sigma(x) = x$, then the variable is atomic and not further expanded.

Definition 41 (Normal Form of String Constraint Formula) *A combined constraint formula φ is any quantifier-free Σ_{Comb}-formula that can be written as a conjunction*

$$\varphi \;=\; \mathcal{E} \;\wedge\; \mathcal{R} \;\wedge\; \mathcal{L} \;\wedge\; \mathcal{C},$$

where

- *$\mathcal{E}$ is a Boolean combination of word equations $s_1 = s_2$;*
- *$\mathcal{R}$ is a Boolean combination of regular-language memberships $x \in \mathcal{L}$ with $\mathcal{L} \in \mathfrak{L}_{\mathrm{Reg}}$;*
- *$\mathcal{L}$ is a Boolean combination of length constraints from $\mathcal{T}_{\mathsf{Str}}$, interpreted in $\mathcal{T}_{\mathrm{LIA}}$ (meaning any linear arithmetics formulae containing lengths of strings as variables);*
- *$\mathcal{C}$ is a Boolean combination of conversion constraints using `to_int`, `from_int`, `to_real`, `from_real`, `to_code` or/and `from_code`.*

Although each category can be solved in isolation, real formulas almost always mix them, which quickly raises the complexity of the decision problem. Solvers therefore support various restricted fragments (e.g. acyclic, flat, or bounded-occurrence).

Stable solution

Because, as it will be described in another chapter, Z3-Noodler uses a stabilisation procedure for searching so-called „stable solution," which we will define right here. First we have to define three key properties: **nonemptiness** of `Lang`, **flatness** of σ and **model completeness**.

Definition 42 (Nonemptiness of `Lang`) *`Lang` is nonempty $\overset{def}{\Leftrightarrow} \forall x \in \mathcal{V}_{\mathsf{Str}} : \mathtt{Lang}(x) \neq \emptyset$*

Definition 43 (Flatness of σ) *σ is flat $\overset{def}{\Leftrightarrow} \forall x_i \in \sigma(x) : \sigma(x_i) = x_i$, i. e. variables representing parts of the string are atomic (not composed of other parts).*

Definition 44 (Model Completeness) *If assignment $\nu : \mathcal{V}_{\mathsf{Str}} \to \Sigma^*$ satisfies:*

- *$\nu(x) \in \mathtt{Lang}(x)$ and*
- *$\nu(x) = \nu(\sigma(x))$ for each $x \in \mathcal{V}_{\mathsf{Str}}$*

Regular Constraints

Regular constraints are a way to impose linguistic constraints on string variables expressed in a regular language. In other words, each variable is understood as an element of a language defined by a regular expression or finite automaton, and its specific assignment is required to respect that language.

Definition 45 (Regular Constraint) *Let $x \in \mathcal{V}_{\mathsf{Str}}$ be a string variable and $\mathcal{R}$ be a regular language over the alphabet Σ. The regular constraint has the form:*

$$x \in \mathcal{R}$$

The assignment $\nu(x) \in \Sigma^*$ is a model if and only if $\nu(x) \in \mathcal{L}(\mathcal{R})$, i.e., the string x belongs to the language $\mathcal{R}$.

$\mathcal{R}$ can be represented by a regular expression, a finite-state automaton, or a recursive function, depending on the solver.

We also need to apply logical operations to regular expressions, in such cases, operations are performed inside the solvers that facilitate the solution of the formula. If the conjunction $x \in \mathcal{R}_1 \wedge x \in \mathcal{R}_2$ occurs in the formula, the intersection of languages (e.g. automata) is performed, and the language of the string is restricted to $\mathcal{L}(x) = \mathcal{L}(\mathcal{R}_1) \cap \mathcal{L}(\mathcal{R}_2) \ni x$. Similarly, for the disjunction $x \in \mathcal{R}_1 \vee x \in \mathcal{R}_2$ or the negation $x \notin \mathcal{R}_1$, we modify the language $\mathcal{L}(x)$ to $\mathcal{L}(x) = \mathcal{L}(\mathcal{R}_1) \cup \mathcal{L}(\mathcal{R}_2)$ or $\mathcal{L}(x) = \overline{\mathcal{L}(\mathcal{R}_1)}$ respectively.

Conversion constraints

Many modern string solvers (as defined in the SMT-LIB [2] standard's `String` theory extensions) support conversion functions between strings and numeric domains [8] [3] (as well as between integer and real arithmetics). These conversions are especially relevant in program analysis tasks, where textual inputs are parsed into numbers, or numbers are serialized into textual forms (e.g. printing integers as ASCII digits). Below are some of the most common conversion functions:

`to_int`(x) Interprets the string x as a decimal integer. If x is a valid positive decimal representation, the function yields the corresponding integer. For example, `to_int`$('123') = 123$, `to_int`$('0003') = 3$. If x does not represent a valid integer, `to_int`(x) is often defined to return some error code or a special „undefined“ value (some solvers represent this as -1 or throw a constraint violation).

`from_int`(n) Converts the integer n (an element of $\mathbb{Z}$) into its canonical decimal string representation (e.g., `from_int`$(42) =' 42'$, `from_int`$(-7) = \varepsilon$). Negative numbers are converted to an empty string.

`to_real`(x) **and `from_real`**(r) Until recently, most SMT or string solvers did not support direct parsing of real-valued strings. Common conversions, like `to_int` or `from_int`, were usually the farthest extent to which numeric/string integration was addressed. However, in our latest extension to Z3-Noodler, we introduce string–real conversions: `to_real`(x) interprets a string x as a decimal real, while `from_real(r)` serializes a rational value r into a decimal string.

Semantics

1. `to_real`(x) attempts to parse the string $x \in \Sigma^*$ as a floating-point decimal, i.e., a mandatory integer part, a decimal point, and a fractional part. For example, „3.14“ or „0.001“. If x does not meet the required decimal pattern, the conversion is regarded as „failed,“ returning -1 as a special marker.

2. `from_real`(r) produces a (finite) decimal string. Crucially, if r is negative or is an irrational or periodic real that would require infinite decimal expansion, the empty string (ε) is returned. In other words, if r is out of scope (e.g. negative, or not representable in a straightforward decimal form), `from_real`(r) becomes ε.

Chapter 3

Tools and State of the Art

In the context of software verification and in general formal program verification, a number of solvers have appeared in recent years that try to cover the area of string constraints. These include, for example, Ostrich, cvc4/5, Z3, Z3-Trau or Z3-Noodler. Some solvers function as completely independent string solvers, while others are parts of larger SMT solvers that support other theories, such as linear arithmetic, bit vectors, and others.

3.1 Ostrich

The string solver Ostrich is classified as a specialized string solver, as it concentrates on equations and constraints over strings, including extended operations such as concatenation, substrings or substring replacement. However, Ostrich can be integrated into a wider SMT context, when we integrate it as a module for wider SMT solvers, which only solve string subproblems for it.

In addition, Ostrich can also work with so-called „straight-line“ constraints. This usually means constraints like „string x consists of a concatenation of symbolic variables and constant fragments“, „x is a substring of y“, „$x = \texttt{Replace}(y, \dots)$“, etc. For many such operations, it is important that the solver understands not only logical equality ($x = y$), but also other predicates (for example, x contained in a regular language, x starts with a prefix, x ends with a suffix, etc.). Ostrich solves this flexibility largely by combining so-called derivational techniques for regular expressions and other automaton transformations.

Ostrich can basically deal with linear (or simple) length constraints: it represents the variable length of a string as an integer variable and creates the corresponding linear conditions. If we limit ourselves mainly to relatively simple length constraints, they can be solved in combination with its internal representation of strings. In case more advanced arithmetic is needed (for example, non-linear constraints or complex modules), Ostrich can cooperate with other tools, or for such demanding features, a solver that supports multiple theories can be used (for example, some version of Z3 or CVC4 with modules for strings and arithmetic). In practice, Ostrich is often either deployed as a standalone solver (if the task is well covered by its capabilities), or integrated into SMT solvers as a partial „backend“ for the string part.

One of the key advantages of Ostrich-type solvers is rich support for regular expressions. Many string solvers offer the operation „x belongs to a regular language $\mathcal{L}$“, but their internal mechanics may differ. In the case of Ostrich, the basis is in automata representation and the application of so-called antichain techniques, deterministic algorithms or

derivatives for regular expressions. This means that when the predicate „$x \in \mathcal{R}$“ appears, where $\mathcal{R}$ is a regular expression (for example, `[a-zA-Z0-9]*` or a more complex combination), OSTRICH creates the appropriate finite automaton and gradually narrows it down according to other conditions, until it either finds a satisfactory assignment of symbolic string variables or proves that none exists. From the user's point of view, very detailed conditions can be expressed in the form of strings, and the solver then tries to combine them with other equalities, substrings, substitutions, and the like.

Because string theories are generally computationally demanding, since they work with potentially infinite languages (a string can have any length). OSTRICH uses optimizations such as CEGAR (Counterexample-Guided Abstraction Refinement), specialized data structures for representing and reducing automata, and also a number of heuristics that are intended to speed up the solution.

3.2 CVC5

CVC5 is a modern SMT solver (Satisfiability Modulo Theories) following the CVC (Cooperating Validity Checker) solver line, CVC Lite, CVC3 and subsequently CVC4. The latest development resulted in the CVC5 version (released in 2021), which serves as a successor and continues the development of functionalities started in CVC4. Both solvers are open-source and developed by teams mainly from universities (Stanford, University of Iowa, University of Waterloo) and industrial partners.

CVC5 supports a number of theories: from linear and nonlinear arithmetic (including real and integer number theory) to bit vectors, arrays and data types, to string theory, which is essential for many modern applications. In the area of string solving (i.e. solving constraints over strings), CVC5 is among the most advanced tools.

CVC4 (released around 2011) was designed as a completely new implementation of the solver, following on from the older CVC3. During development, it gradually acquired specialized modules (T-solvers) for an increasingly wider range of theories, including advanced string functions.

CVC5 (released in 2021) is a logical successor, created with the aim of improving the architecture, performance, and maintainability of the code. It contains a completely redesigned core for some theories (including strings). While many features of CVC4 are preserved in CVC5, many optimizations, new heuristics, and a cleaner implementation have been made.

In the context of string solving, it is significant that the string module has evolved continuously – from the initial implementation of basic string operations to support for more advanced functions such as indexOf, replace, substring, regex membership (RegexIn) and, last but not least, more robust combinations with linear arithmetic.

CVC series solvers are among the general SMT solvers using the DPLL(T) approach. For strings, it knows formulations like: $x = \text{Concat}(y, z)$, $\text{Length}(x) = \text{Length}(y) + 3$ či $\text{RegexIn}(z, [0-9]+)$.

From a string theory perspective, it implements concatenation, length, substring, indexof, replace, regex (RegexIn, RegexConcat, RegexUnion, RegexStar).

Regarding the combination of string solving with arithmetic theory solving, CVC5 supports length constraints and ensures higher efficiency and code modularity (than previous CVC solvers). Various subsystems were unified and the integration of new theories was simplified.

CVC5 also supports direct conversions between integers and strings using the function symbols `str.to_int` and `int.to_str`. In CVC5, support for these conversion functions is implemented through a decision procedure for converting between strings and character code points. This procedure allows for efficient processing of operations such as case conversions and is integrated into the string solver. It is important to note that the `str.to_int` function is partial and is not defined for strings that do not correspond to a valid numeric notation. This means that when attempting to convert a non-numeric string to an integer, the solver will return an undefined value or an error status.

3.3 Kaluza

Kaluza stands out as one of the earliest specialized string solvers, widely recognized as a pioneering effort in tackling constraints over unbounded strings. Developed at Stanford University between 2010 and 2012, Kaluza heavily influenced subsequent work in software verification, security analysis, and various forms of static analysis focused on string-manipulating programs. Despite its discontinuation, it still holds historical importance due to the conceptual groundwork it laid and its lasting impact on how string constraints are approached today.

In essence, Kaluza works by translating string constraints into a combination of integer and bit-level constraints. For each string variable x, the solver maintains an integer variable that represents $|x|$, the string's length. This mapping facilitates handling length-based conditions, such as $|x| > 5$ or $|x| = |y|$. The actual content of x is then represented as a fixed-size array of 8-bit (ASCII) characters with a length that matches $|x|$. Operations like equality, concatenation, or substring are thus captured through constraints on these character arrays. For instance, if x is defined as $\texttt{Concat}(y, z)$, Kaluza enforces $|x| = |y| + |z|$ and partitions the array for x into segments that come from y and z respectively. Meanwhile, more specialized functions like $\texttt{substring}(x, i, \ell)$ or $\texttt{indexOf}(x, y)$ are expanded symbolically so that the resulting relationships can be processed as bitwise constraints and integer inequalities (e.g., valid index ranges).

Kaluza 's arrival in the field was significant because it addressed nontrivial string operations such as substringsubstring, indexOfindexOf, and replacereplace, along with regular expression constraints, at a time when the main SMT solvers mostly lacked native support for strings. The solver's public benchmark suite, which included cases from JavaScript programs, HTML form validation routines, and other web-application settings, soon became a standard reference for newer string solvers (such as Z3-str, S3, or CVC5) to measure their performance and coverage. Furthermore, the fundamental idea of pairing each string variable with a length variable and a bit-vector array has been adopted and refined by many later tools, forming a conceptual backbone for modern string constraint solving.

Nonetheless, Kaluza faces several limitations in practice. Because each character in a string is modeled as an 8-bit element, the solver can encounter serious scalability issues, especially for problems involving large or unbounded string lengths. This bit-vector approach often leads to an exponential blow-up in array constraints, which can exhaust memory and computational resources. Its arithmetic integration is likewise relatively simplistic, relying on manual bindings between array indices and integer variables; in comparison, current SMT solvers such as CVC5 or Z3 variants have more cohesive frameworks where string constraints and integer (or even non-linear) constraints can interact more fluidly through techniques like DPLL(T). Another drawback is that Kaluza is no longer under active development, leaving certain features—like advanced string functions or more comprehen-

sive handling of edge cases—incomplete. Its partial support for regular expressions, which sometimes depends on enumerating potential matches, can also lead to timeouts with more complex patterns or very large input sizes. By contrast, solvers like OSTRICH or Z3-NOODLER employ automata-based algorithms and derivatives, handling regex constraints more efficiently.

Despite these drawbacks, the role of KALUZA in shaping the trajectory of string solver research is undeniable. It demonstrated early on that symbolic reasoning about strings could be incorporated into automated program analysis, which paved the way for the more robust and specialized tools we see today. Even though new solvers have surpassed KALUZA in terms of performance and feature sets, the historical and conceptual influence of KALUZA continues to resonate within the string constraint solving community.

3.4 Z3

The basic string solver in Z3 is part of its standard theory library and is designed as a modular component that allows solving equations and constraints over strings within the broader SMT architecture. The string module fits into the classic DPLL(T) framework, so it can be freely combined with theories such as linear arithmetic, bit-vectors, arrays, or other custom theories.

Z3 offers built-in support for string constants and dedicated solving of constraints using strings, sequences, and regular expressions. In particular, it implements the following core operations:

- **Concatenation (`Concat(x,y)`)**: Joins two strings by appending `y` to the end of `x`, yielding a new string $\texttt{x} \,\|\, \texttt{y}$.
- **Equality / Inequality ($x = y$, $x \neq y$)**: Tests whether two strings have exactly the same sequence of characters (or not).
- **Length (`Length(x)`)**: Returns the number of characters in `x` as an integer.
- **Substring (`Substring(x, start, len)`)**: Extracts the contiguous substring of `x` beginning at index `start` (0-based) of exactly `len` characters.
- **IndexOf (`IndexOf(x,y)`)**: Returns the first position (0 based) where the substring `y` appears in `x` or -1 if `y` is not found.
- **Replace (`Replace(x,y,z)`)**: Produces a new string where every (nonoverlapping) occurrence of `y` in `x` is replaced by `z`.
- **Regular-language membership (`RegexIn(x,R)`)**: Checks whether `x` belongs to the regular language described by `R`, with full support for the union, concatenation, and Kleene-star operators.

Unlike some external solvers (e.g. KALUZA), Z3 uses an internal representation of strings and their operations, without conversion to bit-vectors. This allows for efficient combination with the rest of SMT theory, but at the same time it means that its default string solver is strictly focused on deterministic solutions of common string operations, not e.g. advanced techniques over automata or transducers.

3.4.1 String-Number Conversions

Z3 supports the basic conversions `str.to_int(s)` in its default implementation it attempts to convert the string s to an integer (in decimal) and `int.to_str(i)` it converts the integer i to a string. The function `str.to_int` is partial, which means that it is defined only for strings corresponding to a valid number, e.g. „123“, „-45“. In other cases, it returns an undefined value (in practice usually `-1` or a special notation for „undef“).

Z3 supports combinations of theories (strings + arithmetic), but in the basic string solver there is a weaker interaction between theories – for example, information about string lengths is not always efficiently propagated back from arithmetic to the string module. This can lead to inefficient solutions or the need for a lot of backtracking.

These limitations are attempted to be removed by specialized extensions such as Z3-Noodler or Z3-Trau, which integrate more advanced string techniques into Z3 (e.g. working with automata representations or stabilization procedures).

3.5 S3

Solver S3 (Symbolic String Solver) is a specialized SMT solver focused on vulnerability analysis in web applications. It was designed primarily for use in the context of dynamic symbolic execution (DSE), where it is necessary to solve complex combinations of string and arithmetic constraints, even in the presence of regular expressions.

One of the main features of S3 is the ability to work with unbounded strings, i.e., without a predetermined upper limit on their length, while supporting advanced string operations such as `replaceAll`, `substring`, `indexOf`, `length`, and especially membership in regular languages (`RegexIn`). These capabilities are essential when analyzing web applications (e.g. in JavaScript), where regular expressions and length conditions are common.

Unlike some previous approaches (e.g., the Kaluza solver), S3 does not convert string variables to bit arrays or enumerate all possible lengths. Instead, it uses the so-called recursively defined functions that represent repetition (e.g. Kleene's star) and complex operations such as replaceAll. These functions are evaluated lazily (lazy unfolding), i.e. only when necessary to proceed with the calculation.

Internally, S3 is built as an extension of the Z3 solver and specifically its Z3-str component. It uses the same basic SMT framework as Z3, but extends it with the Z3-str-? module, which, unlike the original Z3-str, supports the interaction between string theory and arithmetic theory (including direct feedback between string lengths and arithmetic constraints), handles membership in regular languages even with unlimited repetition (Kleene star), and allows the expression and solution of operations such as `replaceAll` or match using custom recursive rules. Thanks to this expressiveness, S3 is significantly more robust and efficient than previous approaches – when tested on real benchmark sets from the field of web applications, it was able to provide a definitive answer in a much larger number of cases (99.6% compared to about 30% for Kaluza), be up to 20x faster than Kaluza in satisfiable cases, and compared to Z3-str, provide correct models where Z3-str fails due to the lack of propagation of length information back to string theory.

S3 Thus brings a new level of combined support for string and arithmetic theories, with an emphasis on practical use in finding security flaws in web application code.

S3 also supports basic conversion functions between strings and integer values. Specifically, these are the `str.to_int` and `int.to_str` functions, which are essential in analyzing

web applications, where input data is often converted between text and numeric representation (e.g. in forms, parsing parameters from URLs, etc.).

The function `str.to_int`, which is used to convert a string to an integer, is defined in the S3 solver only if the input string corresponds to a valid integer representation (e.g. „123“, „-45“). If the input string does not correspond to any valid number, the result is undefined, so S3 respects this partiality. The solver can thus detect not only cases where the conversion is possible, but also situations where the conversion fails, which is crucial, for example, when detecting insufficient input processing.

Unlike some other solvers, S3 handles this combination incrementally and stably, i.e., the individual components (string theory, arithmetic, conversion) exchange information so that conflicts or satisfiability can be detected as early as possible, without the need to fix the lengths or domains of variables in advance.

3.6 Z3-Trau

Z3-Trau is an extension of the Z3 SMT solver, focused on efficient string constraint solving with an emphasis on realism and stability when processing operations commonly used in practice – such as various variants of `replace`, the `indexOf`, `lastIndexOf`, `substring` functions or regular expressions. Compared to the basic string solver in Z3, Z3-Trau brings an improved architecture with a richer set of heuristics, deeper integration between string theory and arithmetic and, above all, a stabilization mechanism that helps to find consistent solutions even for complex combinations of conditions.

Z3-Trau is built on the principle of string solving with theory support and works within the standard DPLL(T) framework, so it can solve combined formulas containing both string and numeric or Boolean parts. Its key benefit, however, is the way it handles specific string operations that are either not supported at all or supported to a very limited extent in classic solvers (for example, `replaceAll`, or more complex variants of `indexOf` with repetition and indentation).

Z3-Trau uses so-called pattern-aware unrolling – a technique that enables efficient unrolling and evaluation of string operations where patterns in data are worked on. For example, in the operation `replace(x, y, z)` it is important not only to find the occurrence of the substring `y` in `x`, but also to correctly replace this occurrence and update the related arithmetic and logical conditions. Unrolling is performed adaptively here – that is, only to the extent that is necessary to detect conflicts or find a satisfactory assignment. Thanks to this, there is no unnecessary explosion of possible combinations.

One of the important features of Z3-Trau is also the stabilization of the solution. When working with symbolic strings, fluctuations can occur during the evaluation because new information (e.g. about the length of the strings or their positions) can retroactively affect previously created models. Therefore, Z3-Trau uses its own stabilization mechanism, where the representation of individual string variables and their relationships to other parts of the formula is iteratively refined until consistency is achieved. This significantly reduces the need for backtracking, and the solver can reach a decision on satisfiability more quickly.

The solver works with so-called flat string constraints (or with a limited form of straight-line programs), but its strength is in handling real-world cases, for example, from the web application environment or input validation testing. It can also handle combinations of multiple occurrences of the same pattern, partial overlaps and string transformations, which makes it suitable for static analysis, fuzzing or automatic test generation.

Z3-Trau also supports limited work with regular expressions, although in this respect it relies more on syntactic manipulation than on explicit work with automata such as Z3-Noodler. In the case of more complex regex operations, it is recommended to combine the solver with tools that have a more robust infrastructure for working with automata.

Regarding the support of conversions between strings and numbers, Z3-Trau follows the basic implementation of Z3: it has the functions `str.to_int` and `int.to_str`, while respecting their partial definition – i.e., that for example `str.to_int` is not defined for strings that do not correspond to a numerical representation. This feature is reflected in the analysis of conditions, where it may be important to detect invalid conversions as input processing errors.

In benchmark tests where string operations are a key element (e.g., input validation, regex and indexOf combinations, or replaceAll in iterated structures), Z3-Trau outperforms the basic Z3 string solver – both in terms of computation time and solution completeness. It is therefore suitable for use in scenarios where practical cases with realistic patterns need to be handled and where classical models fail due to a lack of interaction between theories.

3.7 Z3-Noodler

Z3-Noodler is a specialized string solver built on top of the Z3 SMT framework. Although it remains compatible with the standard DPLL(T) architecture of Z3, it substantially extends native string theory with advanced algorithms for handling regular constraints, string-integer conversions, and an iterative stabilization-based procedure designed to solve even complex string constraints efficiently. The name „Noodler“ alludes to its internal operation of noodlification [8], which systematically aligns or „noodles through“ equations and intersections of regular languages. Z3-Noodler's approach draws on powerful automata representations while still leveraging Z3's core capabilities for linear arithmetic and Boolean reasoning.

3.7.1 Motivation and History

Although Z3 has long provided a string theory module, the default implementation is relatively basic; it does not always handle advanced operations (e.g., certain `replace` modes, tricky combinations of `indexOf`, or robust finite-automaton-based membership constraints) as efficiently as specialized solvers. Moreover, while Z3 supports linear arithmetic over string lengths and partial string-integer conversions (e.g., `str.to_int`), those features can be incomplete or slow in the presence of large or unbounded strings.

Z3-Noodler arose in response to these limitations. By tightly integrating a well-studied automata-based methodology and an iterative procedure known as stabilization, it strengthens Z3 's ability to handle equations, inequalities, and membership constraints over strings.

3.7.2 Automata-Centric Architecture

Like Ostrich and other automaton-based string solvers, Z3-Noodler represents each string variable by a finite automaton (NFA) (potentially nondeterministic). These NFAs capture the language of possible strings for each variable, which can be intersected with regular expressions of constraints such as `x in [a-z]*` or `RegexIn(x, R)`. Furthermore, it

systematically refines these NFAs during the solving process, discarding infeasible paths or states whenever a contradiction is found in the higher-level logic.

3.7.3 Stabilization-Based Solving

A major hallmark of Z3-Noodler is its stabilization-based procedure for word equations. This procedure operates under the idea that each string variable can be systematically split or concatenated in ways consistent with the constraints until a fixed point (or „stable solution“) is reached. Once no further refinement is possible, the solver extracts a final integer arithmetic formula that captures the lengths of each variable and the numeric relations induced by the string constraints. There are two important concepts we are going to describe now: noodlification and stability.

Noodlification. When Z3-Noodler encounters a string equation, such as $x \cdot y = z$, it attempts to find valid alignments (also called „splits“) between these variables that do not produce empty languages in their corresponding automata. This process is referred to as noodlification: the solver enumerates ways to partition x and y so that it can match them with the partitioned form of z. Each consistent alignment refines the automata involved, possibly introducing new fresh variables for partial segments.

Stable Solutions. Over multiple alignments and intersections, the automaton of each variable becomes more precise, which is done by removing words that are not allowed by the constraints and merging or rewriting the states for feasible paths. Eventually, if the formula is satisfiable, the solver reaches a point where no new conflicts or refinements appear, meaning that the language assignments for each variable have „stabilized“. This stable arrangement effectively encapsulates all feasible string assignments that satisfy the equation-level constraints (and any membership constraints in the form $x \in R$).

At stability, the solver translates all the discovered constraints into a linear arithmetic formula (LIA for integer-string conversions) that relates string lengths, indexing positions, or offset constraints. This formula is then passed on to the arithmetic engine of Z3 for the final verification of satisfiability. If the arithmetic formula is satisfiable, Z3 can produce numeric assignments of lengths, and the solver can back out a consistent mapping from variables to concrete strings. If it is unsatisfiable, Z3-Noodler must backtrack and try different alignments.

3.7.4 Advanced String Operations and Conversions

Although the stabilization-based procedure was originally designed for word equations plus regular constraints, Z3-Noodler extends it to handle the complex operations that frequently arise in code:

- **`indexOf`, `substring`, `replace`**: In real programs, substring manipulations or the search for partial patterns are common. Z3-Noodler can reason about them by introducing constraints on lengths and positions and ensuring that the relevant automata reflect these cuts or merges.
- **`RegexIn` with intersection** of the greater expressions: Because the solver is automaton-based, it can handle the intersection of `Lang`(x) with any user-provided regular expression. Using partial determinization or antichain methods, it can mitigate the state explosion that naive expansions might cause.

- **String-Integer Conversions**: Many verification problems revolve around user input that is textual but interpreted numerically (e.g., form fields that must be numeric). By introducing constraints such as $k = \texttt{to\_int}(x)$, one can unify the arithmetic conditions ($k > 100$) with the string domain (x being purely digits). Z3-Noodler systematically handles such conversions once the stable solution is found, mapping them into a linear arithmetic formula that enumerates possible digit-based intervals.
- **Integer-String Conversions**: Similarly, constraints such as $x = \texttt{from\_int}(k)$ are handled. If k is negative, the string is empty or undefined; if k is non-negative, the solver enumerates which digit strings in $\texttt{Lang}(x)$ can represent k. To avoid enumerating huge swaths of numbers (such as a million-digit string), it uses an underapproximation parameter or merges digits into intervals for large sets of possible numeric values.

Handling Conversions in Detail

Recent work [8] extends Z3-Noodler by embedding the conversions `to_int` and `from_int` into its stabilization framework. The high-level workflow is the following.

1. Solve the equation and membership constraints first via noodlification until a stable solution emerges. This produces a map $\texttt{Lang}(v)$ for each variable v.
2. For each conversion, carefully build a linear arithmetic subformula describing the numeric relationships implied by that conversion. Z3-Noodler merges large sets of strings into intervals to keep the size of the formula tractable. For instance, if $\texttt{Lang}(x)$ has all 2- and 3-digit decimal strings, that covers the intervals $\langle 0, 99\rangle$ and $\langle 100, 999\rangle$.
3. Combine these sub-formulas with the length constraints gleaned from the stable solution. The resulting LIA formula is typically efficient for Z3's integer solver to decide.
4. Backtrack if unsatisfiable. If no arithmetic assignment satisfies the subformula, the solver tries alternative splits or merges from the noodlification stage or eventually reports `unsatisfiable`.

In practical cases, enumerating intervals or rewriting partial numeric constraints is usually enough to represent complex numeric conversions. If, however, $\texttt{Lang}(x)$ is infinite in the digit dimension, Z3-Noodler can underapproximate (e.g., bounding $|x|$ by some small integer) and possibly declare „unknown“ if no assignment is found within that bound.

Chapter 4

Solving String-Float Conversions

String-float conversions (sometimes referred to as „parsing real numbers“ from text, or conversely “stringifying” real numbers) are crucial in many real-world verification tasks. In typical program code, strings derived from user input may represent decimal numbers with fractional parts (e.g., “3.14” or “42.0”). Symbolic constraints arise when an analyzer (like a symbolic execution engine or an SMT-based static checker) needs to track both:

1. **Textual properties** of the string (its characters, length, match to a numeric regular expression, etc.), and
2. **Arithmetic properties** of the interpreted numeric value, e.g., bounding real variables by intervals or imposing comparison constraints.

This chapter formalizes in detail the conversion of strings representing numeric literals to real values in the context of string constraint solving. We assume that each string variable is constrained by a regular expression and the conversion function `to_real` must be converted to a pure linear-arithmetic (LRA) formulation in order to use modern SMT solvers. We also assume, every language is finite, and in case of need, we force it to be finite.

The goal is to replace each atom $k = \texttt{to\_real}(x)$ (or $x = \texttt{from\_real}(k)$) in the initial mixed formula with an equivalent pure LRA formula $\varphi_{k=\texttt{to\_real}(x)}$ ($\varphi_{x=\texttt{from\_real}(k)}$ respectively). Below, we first introduce the detailed notation and motivation, then describe the decomposition procedure into valid and invalid inputs, and finally define interval encoding.

When we start solving for conversion atoms, Z3-Noodler pre-solves the conjunction $R \wedge E$ from definition 41, which results in the pair $(\texttt{Lang}, \sigma)$ (definitions 39 and 40) that is a stable solution section 2.2.3.

4.1 Basic Notation

In this section, we will introduce some basic symbols/predicates/functions used throughout this section.

4.1.1 Solving Basics

Conversion atom. Let us assume that in the formula $\psi = \mathcal{E} \wedge \mathcal{R} \wedge \mathcal{L} \wedge \mathcal{C}$ there is a conversion atom in the $\mathcal{C}$ part:

$$r = \texttt{to\_real}(x)$$

or

$$x = \texttt{from\_real}(r)$$

where k is an arithmetic (real) variable and the atom converts the string x to the corresponding real value or vice versa.

We are going to show how, after solving the $\mathcal{E} \wedge \mathcal{R} \wedge \mathcal{L}$ part, we can construct the $\varphi_{k=\texttt{to\_real}(x)}$ / $\varphi_{x=\texttt{from\_real}(r)}$ formula encoding the conversion atom given. We assume that the languages with which we work are all finite.

Encoding String Parts Properties. We have to define whether parts of the string being converted are decimal or whole, but also whether they are empty or invalid. To encode this, we define the function `dot_position` as follows:

Definition 46 (Dot Position Function) *Let $x_i \in \mathcal{V}_{\mathsf{Str}}$ be a string variable that satisfies $\sigma(x_i) = x_i$ (i.e., atomic). We define the function*

$$\mathit{dot\_position} : \mathcal{V}_{\mathsf{Str}} \to \{-3, -2, -1, 0, 1, 2, \dots\}$$

by

$$\mathit{dot\_position}(x_i) = \begin{cases} \ell_d & \text{when } x_i \text{ is a valid string with a dot} \\ -1 & \text{when } x_i \text{ is a valid string without a dot} \\ -2 & \text{when } x_i \text{ is an invalid string} \\ -3 & \text{when } x_i = \varepsilon \end{cases}$$

where ℓ_d represents the position of the dot from the right side (starting with zero – zero means the string ends with a dot). By valid strings, we mean strings containing only digits, and at most one dot (including the empty string).

4.1.2 Valid and Nonvalid Inputs

To cover all possible values of $x \in \mathcal{V}_{\mathsf{Str}}$, we divide the language $\texttt{Lang}(x)$ into two disjoint components:

$$\texttt{Lang}_x^{\text{valid}} = \texttt{Lang}(x) \cap \left(\{'0', \dots, '9'\}^* \{'.'\} \{'0', \dots, '9'\}^* \cup \{'0', \dots, '9'\}^*\right)$$

$$\texttt{Lang}_x^{\text{non-valid}} = \texttt{Lang}(x) \setminus \left(\{'0', \dots, '9'\}^* \{'.'\} \{'0', \dots, '9'\}^* \cup \{'0', \dots, '9'\}^*\right)$$

- The **valid part** contains strings that actually represent a positive whole or decimal number (more general than the `to_int` conversion), but also the empty string ε.
- The **invalid part** includes the remaining strings – e.g. „+3“, „12a4“.

Then these two parts are composed of a logical disjunction, so that the resulting encoding is complete.

Definition 47 (Length Set) *For a string variable $x \in \mathcal{V}_{\mathsf{Str}}$, let*

$$L_x = \left\{ |w| \;\middle|\; w \in \mathit{Lang}_x^{\mathit{valid}} \right\},$$

be the finite set (because the language itself is finite) of all the word lengths that can occur in $\mathit{Lang}_x^{\mathit{valid}}$.

In the following text, we also use symbols $L_x^{\text{whole}} = \{\, |w| \mid w \in (\texttt{Lang}_x^{\text{valid}} \cap \{'0', \ldots, '9'\}^*)\}$ and $L_x^{\text{decimal}} = \{\, |w| \mid w \in (\texttt{Lang}_x^{\text{valid}} \cap \{'0', \ldots, '9'\}^* \{'.'\} \{'0', \ldots, '9'\}^*)\}$.

Definition 48 (Interval Set) *Let $x \in \mathcal{V}_{\mathsf{Str}}$, $m \in L_x$, and $\ell_d \in \{-1, 0, \ldots, m-1\}$. We define*

$$I_x(m, \ell_d) = \{(\mathsf{low}, \mathsf{high}) \in (\mathit{Lang}_x^{\text{valid}}) \times (\mathit{Lang}_x^{\text{valid}}) \mid P(\mathsf{low}, \mathsf{high})\},$$

where $\mathsf{low}, \mathsf{high} \in \mathbb{N}$. $P(\mathsf{low}, \mathsf{high})$ holds exactly if:

1. $|\mathsf{low}| = |\mathsf{high}| = m$.
2. *The decimal point in* low *and* high *occurs at position ℓ_d (and if $\ell_d = -1$ they do not contain a point).*
3. $\mathit{to\_real}(\mathsf{low}) \leq \mathit{to\_real}(\mathsf{high})$.
4. Maximality: *Let*

$$\delta = \begin{cases} 1, & \ell_d = -1, \\ 10^{-(m-\ell_d-1)}, & \ell_d \geq 0. \end{cases}$$

   Then neither $\mathit{to\_real}(\mathsf{low}) - \delta$ nor $\mathit{to\_real}(\mathsf{high}) + \delta$ is in the set $\{\mathit{to\_real}(w) \mid w \in \mathit{Lang}_x^{\text{valid}}, |w| = m, w_{\ell_d} =' .'\}$.
5. Continuity: *For every $w \in \Sigma^*$ with $|w| = m$ and dot at ℓ_d, if $\mathit{to\_real}(\mathsf{low}) \leq \mathit{to\_real}(w) \leq \mathit{to\_real}(\mathsf{high})$ then $w \in \mathit{Lang}_x^{\text{valid}}$.*

Thus, each (low, high) *encodes a maximal continuous numeric interval* [$\mathit{to\_real}(\mathsf{low})$, $\mathit{to\_real}(\mathsf{high})$] *for words of length m with the specified dot position.*

Because we need to have a set of all string intervals regardless of the position of a dot, we also need the concept of composed interval set, which is defined as follows:

Definition 49 (Composed Interval Set $I_x(m)$) *Let $x \in \mathcal{V}_{\mathsf{Str}}$, $m \in L_x$. We define*

$$I_x(m) = \bigcup_{\ell_d \in \{-1,0,\ldots,m-1\}} I_x(m, \ell_d)$$

In the following sections, we also use symbols $I_x^{\text{whole}}(m)$, which is derived from L_x^{whole} and $I_x^{\text{decimal}}(m)$ from $L_x^{decimal}$.

4.1.3 Special Notation

Number of String Variable Parts. Because the string variable x contained in the conversion atom is divided into parts by $\sigma(x) = x_1, \ldots, x_n$, we use n as a symbol for the number of its parts throughout the text.

New Variables Symbols. In this section, we also use two key symbols:

- `whole_var`
- `decimal_var`

They are fresh variables that represent the whole and the decimal parts as integer numbers (although they are of real type). So, for example, $\texttt{whole\_var}(x_i) = 12 \wedge \texttt{decimal\_var}(x_i) = 345 \Leftrightarrow \texttt{to\_real}(x_i) = 12.345$.

Divided Interval Sets. For each interval $\langle \mathsf{low}, \mathsf{high} \rangle \in I_x(m, \ell)$ containing a dot (that is, $\ell_d \neq -1$), we divide it into integers (or whole parts, not to be confused with whole variant) and decimal parts without a dot. Putting in sets, we get $I_x^{\text{wpart}}(\ell)$ for the whole part and $I_x^{\text{dpart}}(\ell)$ for the decimal part. Given sets are ordered (any ordering, which gives us a bijection between the two sets), so we do not get invalid combinations of the whole and decimal parts.

4.2 Formulae for Conversion

At first, we have to define help formulae for encoding certain properties of string parts, which we do in this section.

4.2.1 Substring Formula for Valid Integer Cases

For encoding parts x_i of the string variable $x \in \mathcal{V}_{\mathsf{Str}}$ from the conversion atom, given by $\sigma(x) = x_1 \dots x_n$, we use the disjunction shown and explained below:

$$\varphi^{\text{valid whole}}_{\texttt{to\_real}(x_i)} \overset{\text{def}}{\Leftrightarrow} \bigvee_{\ell \in L^{\text{whole}}_{x_i}} \left(|x_i| = l \wedge \bigvee_{\langle \mathsf{low}, \mathsf{high} \rangle \in I^{\text{whole}}_{x_i}(\ell)} \chi(x_i, \mathsf{low}, \mathsf{high}) \right)$$

$$\begin{aligned} \chi(x_i, \mathsf{low}, \mathsf{high}) \overset{\text{def}}{\Leftrightarrow} & \texttt{to\_real}(\mathsf{low}) \leq \texttt{to\_real}(x_i) \wedge \\ & \texttt{to\_real}(x_i) \leq \texttt{to\_real}(\mathsf{high}) \wedge \\ & \texttt{is\_int}(x_i) \wedge \\ & \texttt{dot\_position}(x_i) = -1 \end{aligned}$$

does three things:

1. **Length refinement.** Every admissible length ℓ taken from the precalculated set $L^{\text{whole}}_{x_i}$ is considered separately. Because regular languages are closed under the left quotient, the projection on a fixed length preserves regularity and can be computed once in a preprocessing step [9] [19].
2. **Interval enumeration.** For this length, the finite family $I^{\text{whole}}_{x_i}(\ell)$ partitions the language into interval words: each pair $\langle \mathsf{low}, \mathsf{high} \rangle$ gives the lexicographically smallest and largest word of the same length, producing a closed numeric interval after conversion.
3. **Linear guard.** Inside χ we assert:
   - The arithmetic bounds to x_i,
   - the predicate $\texttt{is\_int}(x_i)$ excluding any embedded dot and
   - $\texttt{dot\_position}(x_i) = -1$ which distinguishes plain-integer strings from the decimal and error cases.

4.2.2 Substring Formula for Valid Decimal Cases

The outer disjunction iterates over all possible total lengths $\ell \in L^{\text{decimal}}_{x_i}$. For each length, we split the word at an implicit dot position:

- ℓ_w characters on the whole side,
- one literal dot,
- $\ell = \ell - \ell_w - 1$ characters on the fractional side.

Two interval families: $I_{x_i}^{\text{wpart}}$ and $I_{x_i}^{\text{dpart}}$ are consulted independently, but due to their ordering, we get the original combinations of the decimal strings, so they cover the same set of intervals.

The subformula $\chi(x_i, l_w, h_w, l_d, h_d, \ell_d)$ asserts the integerness of both substrings, clamps them to their respective interval limits, forces the recorded dot position to exactly ℓ_d, and finally defines the real composite value $\texttt{to\_real}(x_i) = \texttt{whole\_var}(x_i) + \frac{\texttt{decimal\_var}(x_i)}{10^{\ell_d}}$.

All arithmetic remains linear because ℓ_d is a compile time constant inside the branch.

$$\varphi^{\text{valid decimal}}_{\texttt{to\_real}(x_i)} \overset{\text{def}}{\Leftrightarrow} \Bigg(\bigvee_{\ell \in L^{\text{decimal}}_{x_i}} |x_i| = \ell \wedge \Bigg(\bigvee_{\ell_w \in \langle 0, \ell - 1 \rangle} \ell_d = \ell - \ell_w - 1 \wedge \Bigg(\bigvee_{\langle l_w, h_w \rangle \in I^{\text{wpart}}_{x_i}(\ell_w), \langle l_d, h_d \rangle \in I^{\text{dpart}}_{x_i}(\ell_d)} \chi(x_i, l_w, h_w, l_d, h_d, \ell_d) \Bigg) \Bigg) \Bigg)$$

$$\begin{aligned} \chi(x_i, l_w, h_w, l_d, h_d, \ell_d) \overset{\text{def}}{\Leftrightarrow} & \texttt{is\_int}(\texttt{whole\_var}(x_i)) \wedge \\ & \texttt{is\_int}(\texttt{decimal\_var}(x_i)) \wedge \\ & l_w \leq \texttt{whole\_var}(x_i) \wedge \\ & \texttt{whole\_var}(x_i) \leq h_w \wedge \\ & l_d \leq \texttt{decimal\_var}(x_i) \wedge \\ & \texttt{decimal\_var}(x_i) \leq h_d \wedge \\ & \texttt{to\_real}(x_i) = \texttt{whole\_var}(x_i) + \frac{\texttt{decimal\_var}(x_i)}{10^{\ell_d}} \wedge \\ & \texttt{dot\_position}(x_i) = \ell_d \end{aligned}$$

The part $x_i = \texttt{whole\_var}(x_i) + \frac{\texttt{decimal\_var}(x_i)}{10^{\ell_d}}$ cannot be encoded directly in the resulting formula due to its excessive complexity (since it is syntactically nonlinear and cannot be expressed in LRA). This is not a real problem, since the value of l_d is known to us. We can therefore calculate the value of $10^{-\ell_d}$ during solving and multiply $\texttt{decimal\_var}(x_i)$ by the resulting constant. The product $\texttt{decimal\_var}(x_i) \cdot 10^{-\ell_d}$ is now linear and accepted by the Z3 solver.

Equivalently, we have to also define cases for non-valid and empty (containing ε) string:

$$\begin{aligned} \varphi^{\text{non-valid}}_{\texttt{to\_real}(x_i)} \overset{\text{def}}{\Leftrightarrow} & \texttt{len}(|x_i|, \texttt{Lang}^{\text{non-valid}}_{x_i}) \wedge \\ & \texttt{to\_real}(x_i) = -1 \wedge \\ & \texttt{dot\_position}(x_i) = -2 \end{aligned}$$

$$\begin{aligned} \varphi^{\text{contains\_epsilon}}_{\texttt{to\_real}(x_i)} \overset{\text{def}}{\Leftrightarrow} & |x_i| = 0 \wedge \\ & \texttt{to\_real}(x_i) = -1 \wedge \\ & \texttt{dot\_position}(x_i) = -3 \end{aligned}$$

4.2.3 Formulae Encoding Number Properties

Dot position function contains not only information about the position of the dot within the string, but also possibly additional knowledge about the validity or emptiness of the string, which is why we encode the variable properties as follows: The formulae operating with dot position:

$$\varphi_{\text{dot\_exists}}(x) \overset{\text{def}}{\Leftrightarrow} \bigvee_{x_i \in \sigma(x)} \texttt{dot\_position}(x_i) \geq 0$$

The formula $\varphi_{\text{dot\_exists}}$ guarantees that at least one segment xi in the decomposition $\sigma(x) = x_i, \ldots, x_n$ includes a decimal point. In many real-number grammars (including our one), the overall string must contain exactly one dot, so this constraint enforces the „at least one“ aspect of that rule and eliminates trivial all-integer concatenations. Because the converted number doesn't have to contain any dot, we will use this one in another section for dividing cases.

$$\varphi_{\text{exists\_non\_epsilon}}(x) \overset{\text{def}}{\Leftrightarrow} \bigvee_{x_i \in \sigma(x)} \texttt{dot\_position}(x_i) \neq -3$$

The formula $\varphi_{\text{exists\_non\_epsilon}}$ forces the overall string not to collapse to the empty word. Combined with length constraints per variable, it prunes an otherwise large class of spurious models where every component is ε.

$$\varphi_{\text{just\_one\_dot}}(x) \overset{\text{def}}{\Leftrightarrow} \bigwedge_{i=1}^{n} \Bigg(\bigwedge_{j=i+1}^{n} \big(\texttt{dot\_position}(x_i) \geq 0 \implies (\texttt{dot\_position}(x_j) = -1 \vee \texttt{dot\_position}(x_j) = -3)\big) \Bigg)$$

$\varphi_{\text{just\_one\_dot}}$ encodes the uniqueness condition. It is a pairwise implication: once some x_i advertises a non-negative dot position, every later x_j must either be an integer ($\texttt{dot\_position}(x_j) = -1$) or be empty ($\texttt{dot\_position}(x_j) = -3$). Asymmetric ordering avoids redundant symmetrical cases and reduces the search space.

$$\varphi_{\text{no\_invalid}}(x) \overset{\text{def}}{\Leftrightarrow} \bigwedge_{x_i \in \sigma(x)} \texttt{dot\_position}(x_i) \neq -2$$

The formula $\varphi_{\text{no\_invalid}}$ guarantees the absence of syntactically corrupted components as soon as the „one-dot“ protocol is in effect. It is useful for applications that reject mixed strings such as $'12a.4'$ early, avoiding expensive arithmetic reasoning on obviously invalid data.

$$\varphi_{\text{dot\_constraints}}(x) \overset{\text{def}}{\Leftrightarrow} \varphi_{\text{exists\_non\_epsilon}}(x) \wedge \varphi_{\text{just\_one\_dot}}(x) \wedge \varphi_{\text{no\_invalid}}(x)$$

The $\varphi_{\text{dot\_constraints}}$ consolidates three dot-related predicates into a single, unified constraint.

4.2.4 Formulae Encoding `to_real`

After defining the helping formulae, we are going to use them to build the final converting formulae.

Valid Case formulae

$$\varphi^{\text{valid}}_{\texttt{to\_real}(x)} \overset{\text{def}}{\Leftrightarrow} \varphi_{\text{dot\_constraints}} \wedge \bigvee_{\substack{\ell_1 \in L_{x_1} \\ \vdots \\ \ell_n \in L_{x_n} \\ \sum_{i=1}^{n} \ell_i \neq 0}} \eta(x, \ell_1, \ldots, \ell_n) \wedge \Big(\big(\neg \varphi_{\text{dot\_exists}}(x) \big) \implies \big(\texttt{to\_real}(x) = \tau_{\text{integer}} \big) \Big)$$

The large disjunction in $\varphi^{\text{valid}}_{\texttt{to\_real}(x)}$ models a multisegment number obtained by concatenating $|n|$ words. Each branch fixes concrete lengths $\ell_1, \ldots, \ell_n$ such that the total length is non-zero. In addition, if no segment contains a point (that is, $\neg\varphi_{\text{dot\_exists}}(x)$), then the entire string is interpreted as a pure integer by τ_{integer}, matching $\texttt{to\_real}(x)$. The helper

$$\eta(x, \ell_1, \ldots, \ell_n) \overset{\text{def}}{\Leftrightarrow} \bigwedge_{i=1}^{n} \big(\texttt{dot\_position}(x_i) \geq 0 \implies \texttt{to\_real}(x) = \tau_{\text{whole}} + \texttt{to\_real}(x_i) + \tau_{\text{decimal}} \big)$$

then stipulates that the unique segment with a dot (say x_i) splits the concatenation into an integer prefix, $x_1 \ldots x_{i-1} \cdot x_i^{\text{whole}}$, and a fractional suffix, $x_i^{\text{dec}} \cdot x_{i+1} \ldots x_n$. Equation $\texttt{to\_real}(x) = \tau_{\text{whole}} + \texttt{to\_real}(x_i) + \tau_{\text{decimal}}$ adds the numeric value of the „pivot“ x_i to two weighted sums that represent the surrounding segments.

Sum Formulae

The term τ_{whole} aggregates every x_j to the left of the dot. Each of these words contributes its numeric value, multiplied by a power of ten whose exponent is the number of still unknown digits on its right, namely all lengths $\ell_{j+1}, \ldots, \ell_{i-1}$ and the integer part of x_i. The factor $\text{sgn}(\ell)$ is defined as follows:

$$\text{sgn}(\ell) = \begin{cases} 1 & \text{when } \ell > 0, \\ 0 & \text{otherwise,} \end{cases}$$

thus seamlessly skips optional segments without branching.

$$\tau_{\text{whole}} \overset{\text{def}}{=} \sum_{j=1}^{i-1} \texttt{to\_real}(x_j) \cdot 10^{\ell_{j+1} + \cdots + \ell_{i-1} + (\ell_i - \texttt{dot\_position}(x_i) - 1)} \cdot \text{sgn}(\ell_j)$$

The term τ_{decimal} plays the symmetric role for the right side of the dot:

$$\tau_{\text{decimal}} \overset{\text{def}}{=} \sum_{j=i+1}^{n} \texttt{to\_real}(x_j) \cdot 10^{-(\ell_{i+1} + \cdots + \ell_j + \texttt{dot\_position}(x_i))} \cdot \text{sgn}(\ell_j).$$

Finally, the pure-integer case is handled by

$$\tau_{\text{integer}} \overset{\text{def}}{=} \sum_{j=1}^{n} \texttt{to\_real}(x_j) \cdot 10^{\ell_{j+1} + \cdots + \ell_n} \cdot \text{sgn}(\ell_j)$$

Invalid Case Formulae

The formula $\varphi^{\text{invalid}}_{\texttt{to\_real}(x)}$ captures the complementary case where the joint string does not satisfy the dot protocol. Instead of enumerating all concrete reasons (multiple dots, malformed digit groups, etc.), we conjoin the simple test $\neg\varphi_{\text{dot\_constraints}}$ with the sentinel assignment $\texttt{to\_real}(x_i) = -1$. This keeps the Boolean structure shallow and delegates the real work to the already computed regular languages of the components.

$$\varphi^{\text{invalid}}_{\texttt{to\_real}(x)} = \neg\varphi_{\text{dot\_constraints}} \wedge \texttt{to\_real}(x_i) = -1$$

Final Formulae

Finally

$$\varphi_{k=\texttt{to\_real}(x)} \overset{\text{def}}{\Leftrightarrow} k = \texttt{to\_real}(x) \wedge \left(\varphi^{\text{valid}}_{\texttt{to\_real}(x)} \vee \varphi^{\text{invalid}}_{\texttt{to\_real}(x)}\right) \wedge \bigwedge_{i=1}^{n} \left(\varphi^{\text{valid}}_{\texttt{to\_real}(x_i)} \vee \varphi^{\text{non-valid}}_{\texttt{to\_real}(x_i)} \vee \varphi^{\text{contains\_epsilon}}_{\texttt{to\_real}(x_i)}\right)$$

replaces the single mixed-theory atom $k = \texttt{to\_real}(x)$ by a purely arithmetic skeleton while preserving soundness (no spurious solutions) and completeness (every genuine solution has a match). The outer equality merely aliases the new arithmetic variable k with the value obtained from the reduction, allowing the rest of the verification condition to refer to k in a theory-uniform way.

4.2.5 Formulae Encoding `x = from_real(r)`

The encoding of the `from_real` atom directly mirrors the `to_real` formula. Rather than constructing a separate conversion logic, we simply reuse the valid and invalid case definitions from $\varphi_{k=\texttt{to\_real}(x)}$ and reverse the roles of the string and the real variable. Specifically, we replace the arithmetic variable k with the known real value r and solve for the structure of x. This yields the following.

$$\varphi_{x=\texttt{from\_real}(r)} \overset{\text{def}}{\Leftrightarrow} \big(x = \texttt{from\_real}(r)\big) \wedge \big(\varphi^{\text{valid}}_{k=\texttt{to\_real}(x)}[k \mapsto r] \vee \varphi^{\text{invalid}}_{k=\texttt{to\_real}(x)}[k \mapsto r]\big).$$

This formulation keeps both cases consistent with the structure of `to_real`, ensuring that the round-trip transformation is sound and complete. All constraints remain in pure linear arithmetic, and no additional logic is required to explicitly filter periodic reals — the decomposition on the string side ensures that only finite decimal encodings are considered.

Chapter 5

Implementation

The algorithm 1 is the main entry point for the overall conversion of all `to_real` and `from_real` atoms to the pure linear arithmetic (LRA) form and informing the caller what kind of precision was used (if the formula cannot be solved precisely, it can underapproximate the result). At the very beginning, it initializes the accumulated formula result to the value „true“ and sets the global variable resPrecision to the highest precision level (PRECISE). It then calls the helper function GetVarsSubstitutedInConversions, which prints the exact set of realSubstVars variables from the existing partial solution that are actually replaced by real values in string conversions. To make the algorithm work efficiently, it also creates an empty table tblValidLen, in which it will store the allowed lengths of individual substrings. The main part then consists of a single call GetFormulaForRealSubstVars(**realSubstVars**, **tblValidLen**), which returns the pair $\langle F, p \rangle$ – where F is the LRA formula containing all the required conversions and p indicates whether it is an exact or approximate version. If F is indeed a nonzero formula, the algorithm conjunctively appends it to the result and, if p indicates a worse than exact approximation, rewrites *resPrecision* to this lower level. This single call produces a complete formal description of all conversions, with no remaining mixed atoms, which is eventually returned in the pair $\langle result, resPrecision \rangle$ to the calling SMT solver.

The algorithm 2 iterates through all the `to_real` and `from_real` conversions stored in the conversions data structure. For each conversion, it takes the corresponding string variable and calls the Solution.GetSubstitutedVars helper function to find out what basic terms (subvariables) are being substituted into this string. Then each variable found is added to the *realSubstVars* set. After this phase is complete, *realSubstVars* contains exactly all the variables that the next step (i.e., the call to GetFormulaForRealSubstVars) will actually need when generating the LRA formulae and returning them to the caller.

The algorithm 3 takes care of the conversion of „small“ conversions (conversion of atomic substrings) that arise by dividing a string variable into several smaller ones into a purely arithmetic formula. It accepts as input a set $\mathcal{V}$ of real variables for which the substitution is performed, and an empty map tbl to store the allowed string lengths. First, it prepares two deterministic automata: a DecimalNumberAutomaton given by algorithm 4, which accepts all correct decimal notations, and its counterpart, which recognizes invalid (incomplete or incorrect) representations. For each variable $v \in \mathcal{V}$ it then loads its current regular language A = Solution.Aut(v) and counts the intersections with both automata. If the intersection with the invalid automaton is not empty, it adds a clause to the resulting for-

Algorithm 1: GetFormulaForConversions

Output: $\langle result, resPrecision \rangle$

```
result ← ⊤
resPrecision ← PRECISE
realSubstVars ← GetVarsSubstitutedInConversions()
tblValidLen ← {}
⟨F, p⟩ ← GetFormulaForRealSubstVars(realSubstVars, tblValidLen)
if F ≠ ∅ then
    result ← result ∧ F
if p ≠ PRECISE then
    resPrecision ← p
foreach conv ∈ conversions do
    if conv.type ∈ {TO_REAL, FROM_REAL} then
        result ← result ∧ (
        GetFormulaForRealConversion(conv.StringVar(), conv.RealVar()))
return ⟨result, resPrecision⟩
```

Algorithm 2: GetVarsSubstitutedInConversions

Output: *realSubstVars* – set of basic terms that substitute a string variable inside any real conversion

```
realSubstVars ← ∅
                                                    // initialise empty set
foreach conv ∈ conversions do
    foreach v ∈ Solution.GetSubstitutedVars(conv.string_var) do
        realSubstVars ← realSubstVars ∪ {v}
return realSubstVars
```

mula that prohibits these incorrect notations. Next, it sorts the valid part of the language by the individual lengths of the string: for each length ℓ that the automaton accepts, it stores ℓ in tbl[v]. Subsequently, for each word interval determined in this way, it creates a set of „interval words“ using GetIntervalWords(A_ℓ) and calls EncodeRealIntervalWords to convert them to a disjunction of precise LRA-constraints (i.e., to a set of linear clauses describing what numerical values these text entries can have). It combines these partial formulas into a single φ_v using logical disjunction. It conjunctively adds each φ_v to the global variable result and monitors whether any automatic steps during processing led to under- or overprecision (done via intersection of possibly infinite language with finite of any legal sign – algorithm 5 removes loops from automata, and encodes length as a state chain); in such a case, it updates the variable precision. In the end, the method returns the pair $\langle result, precision \rangle$, which no longer contains any mixed atoms, but only pure arithmetic clauses describing all valid and invalid (forbidden) cases of text conversions.

The algorithm 6 takes as input a regular automaton and the word length for a specific real variable v and returns two key things:

1. A list of interval words – each „interval word“ is represented as a vector of pairs $[a_i, b_i]$, where a_i and b_i determine the allowed range of the digit at the i-th position of the string. In total, a vector of vectors of pairs is created that covers all combinations of possible character ranges determined by the automaton.

2. LRA formula for interval restrictions – in parallel with generating vectors of pairs, the method constructs a linear-arithmetic disjunction or conjunction of clauses like: $a_i \leq \texttt{to\_int}(s[i]) \leq b_i$ for each character $s[i]$. Thanks to this, the resulting formula reflects exactly in which numeric intervals each substring can move, without having to work with full textual restrictions.

Both parts – the interval word vector and the corresponding arithmetic formula – are returned by the method to the caller, so that they can be added to the main conversion procedures and used to accurately encode all possible values of text variables.

The original Noodler implementation, designed for the `to_int` and `from_int` conversions, used the EncodeIntervalWords method to directly generate the LIA formula from interval words. In our extended version, this method is overloaded (in algorithm 7) as follows: instead of generating a formula, it now just constructs and returns a list of closed integer intervals $\langle L, H\rangle$ corresponding to the input vector of pairs $[a_i, b_i]$. The algorithm starts with a single interval $\langle 0, 0\rangle$ and a variable π that specifies the order of the number, initially with the value $\pi = 1$. Then, for each pair of ranges $[a_i, b_i]$ in the current vector, it creates new intervals $\langle a_i \cdot \pi + L, b_i \cdot \pi + H\rangle$ for each $[L, H]$ from the previous iteration and then multiplies π by ten. By iterating over all positions, a complete list of intervals is eventually created that covers exactly and only the numbers corresponding to the original text interval. This clean list serves as a foundation for EncodeRealIntervalWords, which builds the final LRA formula for real numbers on top of it.

The algorithm 8 creates a purely arithmetic condition that uses the `dot_position` attribute to check the correctness of the text form of all string variables in the set $\mathcal{V}$. First, it constructs the clause dotExists by going through all variables and accumulating the disjunction of the statement $\texttt{dot\_position}(x) \geq 0$, so the result is a test of whether a decimal point actually occurs in any string. This is followed by the clause *existsNonEps*, which similarly verifies that at least one string is not empty: An empty word has a special code -3 in its internal representation.

In the third step, the clause *justOneDot* is constructed. The algorithm traverses all pairs of strings (x_i, x_j) with $i < j$. If the first contains a dot, the second is forced to be either an integer ($\texttt{dot\_position} = -1$) or empty ($\texttt{dot\_position} = -3$). This globally prohibits the occurrence of the second or additional dot. The fourth clause *noInvalid* finally prevents error states by adding a requirement for each variable $\texttt{dot\_position}(x) \neq -2$, which is an internal sentinel for invalid parsing.

The resulting formula φ is a simple conjunction $\mathit{dotExists} \wedge \mathit{existsNonEps} \wedge \mathit{justOneDot} \wedge \mathit{noInvalid}$. By adding this single condition to the other conversion constraints, the solver is ensured that exactly one string with a dot will appear in the model, no invalid states will arise, and the entire string set will not be emptied.

The algorithm 9 creates the final LRA formula, which fully replaces the original atoms `to_real` and `from_real`. It first handles the invalid case; it passes this through a separate

clause. Then it uses algorithm 10, which returns all possible vectors of substring lengths (that is, all the ways in which the text can be decomposed into parts of different lengths). For each such combination, first the conjunction of the conditions $|s_i| = \mathit{length\_case}[i]$ is added to ensure the exact dimensions of each section of the string. Then it inserts two conditional subformulas: one using RealPartsSum, which encodes the real value for all possible dot positions, and the other using IntPartsSum, which covers the case where no decimal point appears in any substring. These two constraints are guarded by conditions over the dot positions and combined into the overall formula. All these sub-blocks (cases) are then combined by disjunction into the variable φ_{valid}. Because φ_{valid} is combined with the result of RealConversionsDot (which enforces exactly one dot and prohibits invalid states), the result is returned as the complete formula Φ. This sequence creates a single LRA disjunction, covering all length and text variants of the input, without any remaining mixed atoms.

The algorithm 11 now receives a fixed dot position from its caller and builds the arithmetic term only for that position. Beginning with the substring that actually contains the dot, it multiplies every block to the left by a positive power of ten that reflects the total length of the blocks already processed, and multiplies every block to the right by the matching negative powers so that those digits contribute fractional values. All such products are summed into one expression, and the algorithm simply returns the equality $r = \mathit{sum}$. The guard $\texttt{dot\_position}(s_i) = k$ is attached by the surrounding procedure, allowing this single core calculation to be reused for every admissible dot position.

Algorithm 3: GetFormulaForRealSubstVars

Input: $\mathcal{V}$ – set of real substituted variables
tbl – map *(to be filled)* : variable $\mapsto$ admissible lengths
Output: $\langle result, precision \rangle$

$result \leftarrow \top$
$precision \leftarrow$ PRECISE
$\mathcal{A}_\checkmark \leftarrow$ DecimalNumberAutomaton()
$\mathcal{A}_\times \leftarrow$ Complement($\mathcal{A}_\checkmark$)
foreach $v \in \mathcal{V}$ **do**
  $tbl[v] \leftarrow \varnothing$
  $\Phi_v \leftarrow \bot$
  $\mathcal{A} \leftarrow$ Solution.Aut(v)
  $\mathcal{A}_v^\checkmark \leftarrow$ Reduce$\big((\mathcal{A} \cap \mathcal{A}_\checkmark)$.Trim()$\big)$
  $\mathcal{A}_v^\times \leftarrow$ Reduce$\big((\mathcal{A} \cap \mathcal{A}_\times)$.Trim()$\big)$
  `/* -- non-valid branch --` */
  if $\mathcal{A}_v^\times \neq \varnothing$ **then**
    $\Phi_v \leftarrow \Phi_v \vee \big($GetLengths($\mathcal{A}_v^\times, v$) $\wedge$ `real_ver`$(v) = -1 \wedge$ `dot_pos`$(v) = -2\big)$
  if $\mathcal{A}_v^\checkmark = \varnothing$ **then**
    $result \leftarrow result \wedge \Phi_v$; **continue**
  `/* --` ε `branch (length 0) --` */
  if $\varepsilon \in \mathcal{L}(\mathcal{A}_v^\checkmark)$ **then**
    $\Phi_v \leftarrow \Phi_v \vee (|v| = 0 \wedge$ `real_ver`$(v) \neq -1 \wedge$ `dot_pos`$(v) = -3)$
    $tbl[v]$.`push`(0)
  `/* -- determine length bound --` */
  if $\mathcal{A}_v^\checkmark$ *is acyclic* **then**
    $\ell_{\max} \leftarrow$ States($\mathcal{A}_v^\checkmark$) $- 1$
  else
    $\ell_{\max} \leftarrow$ `m_underapprox_length`
    $precision \leftarrow$ UNDERAPPROX
  `/* -- iterate concrete lengths 1..`$\ell_{\max}$ `--` */
  for $\ell \leftarrow 1$ **to** $\ell_{\max}$ **do**
    $\mathcal{A}_\ell \leftarrow$ Minimise$\big((\mathcal{A}_v^\checkmark \cap$ DigitOrDotAut(ℓ)$)$.Trim()$\big)$
    if $\mathcal{A}_\ell = \varnothing$ **then**
      continue
    $tbl[v]$.`push`(ℓ)
    $\Phi_v \leftarrow \Phi_v \vee \big(|v| = \ell \wedge$
     EncodeRealIntervalWords(`real_ver`(v), v, GetIntervalWords($\mathcal{A}_\ell$))$\big)$
  $result \leftarrow result \wedge \Phi_v$
return $\langle result, precision \rangle$

Algorithm 4: DecimalNumberAutomaton

Output: $(Q, \Sigma, \delta, q_0, F)$ – the NFA quintuple for decimal numbers

$Q \leftarrow \{q_0, q_1\}$ `// q0: integer-part state; q1: fractional-part state`
$\Sigma \leftarrow$ Unicode
$\delta \leftarrow \emptyset$
for $d \in \{$ `'0'`, …, `'9'` $\}$ **do**
  $\delta \leftarrow \delta \cup \{(q_0, d, q_0), (q_1, d, q_1)\}$
$\delta \leftarrow \delta \cup \{(q_0,$ `'.'`$, q_1)\}$
q_0 is the start state
$F \leftarrow \{q_0, q_1\}$ `// both states accept`
return $(Q, \Sigma, \delta, q_0, F)$

Algorithm 5: DigitOrDotAut

Input: n – target string length

Output: $(Q, \Sigma, \delta, q_0, F)$ – NFA quintuple accepting all strings of length n over digits and '.'

$Q \leftarrow \{q_0, q_1, \ldots, q_n\}$
$\Sigma \leftarrow$ Unicode
$\delta \leftarrow \emptyset$ `// no transitions initially`
for $i \leftarrow 0$ **to** $n - 1$ **do**
  foreach $d \in \{$ `'0'`, …, `'9'` $\}$ **do**
    $\delta \leftarrow \delta \cup \{(q_i, d, q_{i+1})\}$
  $\delta \leftarrow \delta \cup \{(q_i,$ `'.'`$, q_{i+1})\}$
$q_0 \leftarrow 0$
$F \leftarrow \{n\}$ `// only the final state accepts`
return $(Q, \Sigma, \delta, q_0, F)$

Algorithm 6: EncodeRealIntervalWords

Input: v – arithmetic variable storing the real value
o – original string variable
$\mathcal{I}$ – list of *interval words*

Output: Φ – disjunction of LRA clauses encoding all words in $\mathcal{I}$

$\Phi \leftarrow \bot$
foreach $w \in \mathcal{I}$ **do**
  $dot \leftarrow \textsc{Find}(\texttt{'.'}, w)$ `// index or` $|w|$
  if $dot = |w|$ **then**
    `// no decimal point ⇒ integer case`
    $\mathcal{J} \leftarrow \textsc{EncodeIntervalWords}(w)$
    foreach $\langle l, h\rangle \in \mathcal{J}$ **do**
      $\Phi \leftarrow \Phi \vee \big(l \le v \le h \wedge \texttt{is\_int}(v) \wedge \texttt{dot\_pos}(o) = -1\big)$
  else
    `// decimal case`
    $w_{\text{whole}} \leftarrow w[0 \ldots dot-1]$
    $w_{\text{decimal}} \leftarrow w[dot+1 \ldots]$
    $d \leftarrow |w_{\text{decimal}}|$
    $\lambda \leftarrow 10^{-d}$ `// multiplier for fractional part`
    if $w_{whole} = \varepsilon$ **then**
      $\mathcal{J}_{\text{whole}} \leftarrow \{\langle 0, 0\rangle\}$
    else
      $\mathcal{J}_{\text{whole}} \leftarrow \textsc{EncodeIntervalWords}(w_{\text{whole}})$
    if $w_{decimal} = \varepsilon$ **then**
      $\mathcal{J}_{\text{dec}} \leftarrow \{\langle 0, 0\rangle\}$
    else
      $\mathcal{J}_{\text{dec}} \leftarrow \textsc{EncodeIntervalWords}(w_{\text{decimal}})$
    $\langle v_W, v_D\rangle \leftarrow \textsc{PartsOfRealNumber}(v)$ `// fresh int vars`
    for $i \leftarrow 1$ **to** $|\mathcal{J}_{whole}|$ **do**
      $\langle l_W, h_W\rangle \leftarrow \mathcal{J}_{\text{whole}}[i]$
      $\langle l_D, h_D\rangle \leftarrow \mathcal{J}_{\text{dec}}[i]$
      $\Phi \leftarrow \Phi \vee \big(\texttt{is\_int}(v_W) \wedge \texttt{is\_int}(v_D) \wedge l_W \le v_W \le h_W \wedge l_D \le v_D \le h_D \wedge v = v_W + \lambda\, v_D \wedge \texttt{dot\_pos}(o) = dot\big)$
return Φ

Algorithm 7: EncodeIntervalWords

Input: ω – interval word (array of digit-interval pairs, most $\mapsto$ least significant)
Output: Γ – list of closed numeric intervals $\langle low, high \rangle$

$\Gamma \leftarrow [\langle 0, 0 \rangle]$ // start with interval [0,0]
assert $|\omega| > 0$
$\pi \leftarrow 1$ // current place value 10^k
$split \leftarrow \bot$
for $i \leftarrow |\omega| - 1$ **to** 0 **step** -1 **do** // scan least $\rightarrow$ most significant
  $\langle a, b \rangle \leftarrow \omega[i]$
  $a \leftarrow a - \texttt{DIGIT\_START}$; $b \leftarrow b - \texttt{DIGIT\_START}$
  if $split = \bot$ **then** // no branching so far
    $\Gamma[0].low \mathrel{+}= a \cdot \pi$
    $\Gamma[0].high \mathrel{+}= b \cdot \pi$
    if $a \neq 0$ ***or*** $b \neq 9$ **then**
      $split \leftarrow \top$
  else
    // branch on every possible digit
    $\Gamma_{\text{new}} \leftarrow [\,]$
    foreach $\langle L, H \rangle \in \Gamma$ **do**
      for $d \leftarrow a$ **to** b **do**
        $\Gamma_{\text{new}} \leftarrow [\Gamma_{\text{new}} \mid \langle d \cdot \pi + L,\ d \cdot \pi + H \rangle]$
    $\Gamma \leftarrow \Gamma_{\text{new}}$
    $\pi \leftarrow 10 \cdot \pi$
return Γ

Algorithm 8: RealConversionsDot

Input: $\mathcal{V}$ – set of real-substitution variables
Output: Φ – conjunction of dot-protocol constraints

$existsNonEps \leftarrow \bot$
foreach $x \in \mathcal{V}$ **do**
  $existsNonEps \leftarrow existsNonEps \vee \texttt{dot\_position}(x) \neq -3$
$justOneDot \leftarrow \top$
for $i \leftarrow 1$ **to** $|\mathcal{V}|$ **do**
  for $j \leftarrow i+1$ **to** $|\mathcal{V}|$ **do**
    // if x_i has a dot, x_j must be -1 (int) or -3 (ε)
    $justOneDot \leftarrow justOneDot \wedge \big(\big(\texttt{dot\_position}(x_i) \geq 0 \implies$
     $[\texttt{dot\_position}(x_j) = -1 \ \vee \ \texttt{dot\_position}(x_j) = -3]\big)\big)$
$noInvalid \leftarrow \top$
foreach $x \in \mathcal{V}$ **do**
  $noInvalid \leftarrow noInvalid \wedge \texttt{dot\_position}(x) \neq -2$
$\Phi \leftarrow existsNonEps \wedge \ justOneDot \wedge \ noInvalid$
return Φ

Algorithm 9: GetFormulaForRealConversion

Input: s – string variable
  r – real variable
Output: Φ – formula encoding conversion of s into real r

$\delta \leftarrow$ RealConversionsDot($conv$)
$invalid \leftarrow (\neg\delta) \wedge (r = -1)$
$valid \leftarrow (0 \leq r) \wedge \delta$
$\Phi \leftarrow invalid \vee valid$
if $|\sigma(s)| = 0$ **then**
  return Φ
$lengthCases \leftarrow$ AllLengthCombinations($\sigma(s)$)
$\Psi \leftarrow \bot$
foreach $\mathbf{L} \in lengthCases$ **do**
  $\nu \leftarrow \top$
  $\Theta \leftarrow \bot$
  for $j \leftarrow 0$ **to** $|\sigma(s)| - 1$ **do**
    $\nu \leftarrow \nu \wedge (|s_j| = \mathbf{L}[j])$
    for $k \leftarrow 0$ **to** $\mathbf{L}[j] - 1$ **do**
      $\Theta \leftarrow \Theta \vee ((\texttt{dot\_position}(s_j) = k) \implies$ RealPartsSum($\sigma(s), j, k, \mathbf{L}, r$))
  $\Theta \leftarrow \Theta \vee$ IntPartsSum($s, r, \mathbf{L}$)
  $\nu \leftarrow \nu \wedge \Theta$
  $\Psi \leftarrow \Psi \vee \nu$
$valid \leftarrow valid \wedge \Psi$
$\Phi \leftarrow invalid \vee valid$
return Φ

Algorithm 10: AllLengthCombinations

Input: *substVars* – sequence of substituted variables
Output: *lengthCases* – list of length vectors for each substitution

```
1  lengthCases ← {[]}                          // initialize with one empty vector
2  foreach v ∈ substVars do
3      possibleLengths ← v.PossibleLengths()    // retrieve lengths for v
4      newCases ← ∅
5      foreach ℓ ∈ possibleLengths do
6          foreach case ∈ lengthCases do
7              newCase ← case
8              newCase ← [newCase | ℓ]
9              newCases ← newCases ∪ {newCase}
10     lengthCases ← newCases
11 return lengthCases
```

Algorithm 11: RealPartsSum

Input: *subst_vars* – vector of string variables
index – index of the variable containing the dot
dp – position of the dot within that variable
one_case – vector of the lengths of each string part
r – real variable
Output: Φ – formula encoding the sum of real-part contributions at the given dot position

```
1  sum ← to_real(subst_vars[index])
2  place ← 10^dp
3  for j ← index − 1 to 0 step −1 do
4      if one_case[j] = 0 then
5          continue
6      sum ← sum + to_real(subst_vars[j]) · place
7      place ← place · 10^one_case[j]
8  place ← 10^−(one_case[index]−dp−1)
9  for j ← index + 1 to |one_case| − 1 do
10     if one_case[j] = 0 then
11         continue
12     place ← place · 10^−one_case[j]
13     sum ← sum + to_real(subst_vars[j]) · place
14 Φ ← (r = sum)
15 return Φ
```

Algorithm 12: IntPartsSum

Input: s – string variable
r – real variable
one_case – vector of the lengths of the string parts
Output: Φ – formula encoding sum of integer parts

$\Phi \leftarrow \neg\varphi_{\texttt{dot\_exists}}(s)$
$sum \leftarrow 0$
$place \leftarrow 1$
for $i \leftarrow |\sigma(s)| - 1$ **to** 0 ***step*** -1 **do**
    $var \leftarrow s_i$ // $s_i \in \sigma(s)$
    $len \leftarrow one\_case[i]$
    if $len > 0$ **then**
        $sum \leftarrow sum + \texttt{to\_real}(var) \cdot place$
        $place \leftarrow place \cdot 10^{len}$
$\Phi \leftarrow \Phi \wedge \big(r = sum\big)$
return Φ

Chapter 6

Experimental Evaluation

In this chapter, we present a comprehensive empirical evaluation of our extended Z3-Noodler implementation that supports string-float conversions.

6.1 Experimental Setup

We compare three solver configurations on five benchmark suites drawn from the `smt-bench` repository[1]. The first configuration is the unmodified Z3 solver with built-in string-to-integer conversion support. The second configuration uses the original version of Z3-Noodler, which is based on string-integer conversion procedures. The third and final configuration is our enhanced version of Z3-Noodler, which includes native support for string-to-float conversions.

The benchmark suites were selected to cover a representative mix of real-world string processing problems with numeric semantics. In the following, we describe the five suites used (denoted B-F, since suite A contains our own test instances):

B *Numeric Decodings:* decoding a string of digits as letters – 3 instances.

C *IPv4 Generation:* enumerate all valid IPv4 addresses from strings – 594 instances.

D *Abbreviations Expansion:* validate a compressed word – 1 instance.

E *Email Date Parsing:* parse dates in RFC 2822 headers – 21 instances.

F *IP String-to-Numeric:* convert IPv4/IPv6 string representations to internal numeric form – 132 instances.

The experiments were carried out on a machine with an Intel Core i7-10700K CPU running at 3.8 GHz and 16 GB of RAM. All tests were performed on Ubuntu 24.04 using Z3 version 4.8.12 and Z3-Noodler 1.3.0, with Python 3.12.3 orchestrating the benchmarks. We set a timeout of 60 seconds per instance. For each run, we recorded the result (`sat`, `unsat`, or `timeout`) and the total time taken (including timeout cases).

6.2 Results

The table 6.1 summarizes, for each suite and each solver configuration, the counts of `sat`, `unsat`, and `timeout` (TO) outcomes.

[1] https://github.com/VeriFIT/smt-bench

Table 6.1: Outcome counts (`sat`/`unsat`/`timeout`)

Suite	Z3 (int)			Z3-Noodler (int)			Z3-Noodler (float)		
	sat	unsat	TO	sat	unsat	TO	sat	unsat	TO
B	2	1	0	2	1	0	2	1	0
C	24	562	8	31	563	0	31	563	0
D	1	0	0	1	0	0	1	0	0
E	3	18	0	3	18	0	3	18	0
F	93	37	2	94	37	1	94	37	1

Detailed timing results are given in table 6.2.

Table 6.2: Total solving time (s) across all configurations

Suite	Z3 (string-int)	Z3-Noodler (string-int)	Z3-Noodler (string-float)
B	0.51	0.76	2.67
C	825.25	45.09	34.86
D	0.03	0.06	0.06
E	0.74	0.83	0.67
F	343.88	173.15	143.99

Our extended string–float conversion yields substantial performance improvements in most benchmarks while preserving full correctness even in actual `sat` cases. In the IPv4 Generation suite (C), for example, our solver completes all 594 instances in just 34.86 s, 95.8 % faster than Z3's 825.25 s and 22.7 % faster than the original Z3-Noodler 45.09 s, demonstrating that native floating support not only scales, but can outperform the highly optimized string–int pipeline.

In suites where some instances are `sat`, we likewise see clear wins over the int version. Email Date Parsing (E) finishes in 0.67 s versus 0.83 s (19.3 %), and IP string-to-numerical (F) in 143.99 s versus 173.15 s (16.8 %) in 132 instances (including 94 `sat`). Suite D (Abbreviations Expansion) is effectively identical (0.06 s each).

Beyond raw throughput, every `sat`/`unsat` classification perfectly matches the integer baselines: our strict finite-decimal semantics never introduce spurious solutions on B–D and maintain full agreement elsewhere. By combining formal soundness, robust scalability (minimal timeouts), and double-digit speedups against the int converter, our augmented Z3-Noodler proves to be correct and highly performant. These results make a compelling case for integrating string–float conversion support into real-world SMT-based verification and analysis pipelines.

Accounting for the `timeout` answers. Only suites C and F produced any `timeout` results:

- **Z3-Noodler (string–int)** recorded no timeouts in C and one in F; substituting for its isolated solve time yields the total of 173.15 s reported for suite F.
- **Z3 (string–int)** timed out on eight cases in C and two in F; replacing each placeholder with its actual solve time produces 825.25 s for C and 343.88 s for F.

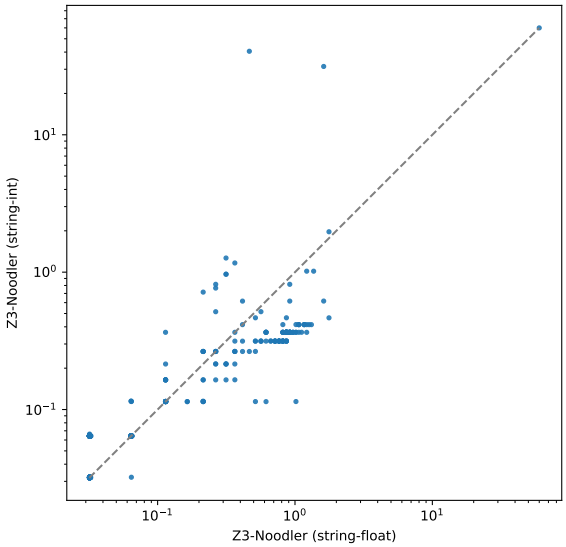

Figure 6.1: Log-log comparison of solving times between Z3-Noodler (string-float) and Z3 (string-int).

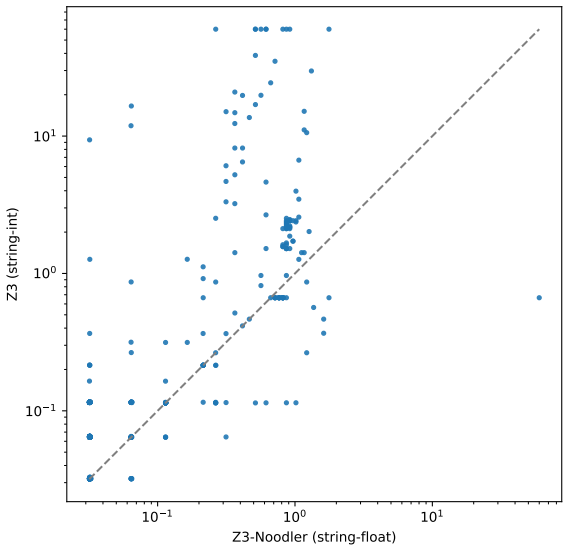

Figure 6.2: Log-log comparison of solving times between Z3-Noodler (string-float) and Z3-Noodler (string-int).

- **Z3-Noodler (string–float)** recorded a single timeout in F; using its isolated 143.99 s solution time gives the final figure for suite F.

No other suites contain `timeouts`, so their reported times already reflect the full solver effort.

Chapter 7

Conclusion

This work is the first to systematically address the precise support of string-to-real conversions in SMT solvers. While previous tools only applied integer conversions, in the real world, decimal notations are an integral part of validators, parsers, and security checks. By extending Z3-Noodler with `to_real`/`from_real`, we have therefore opened up the formal verification of a completely new class of program behaviors, previously covered only by rough approximations or manual interventions.

At the theoretical level, we have moved from pure integer linear arithmetic (LIA) to linear real arithmetic (LRA), thereby obtaining direct support for decimal shifts as constant coefficients (e.g. 10^{-k}) without the need for nonlinear expressions. However, at the same time, we have retained the discrete validation of integer parts using the `is_int` predicate, which combines the greatest advantages of both theories. Although this approach requires special treatment of the two modes, „without dot“ and „with dot“, and generates a slightly larger number of disjunctions for all length vectors, the result is still a purely linear, decidable LRA formula. Thanks to the modular design, we have thus achieved a compromise between formalism and efficiency.

The implementation was seamlessly integrated into the Z3-Noodler module and was evaluated in five public benchmark suites (751 SMT instances in total). Despite the inherent higher combinatorial complexity introduced by native string-to-real conversions, we actually reduced the average solving time compared to the original integer-only conversions of Z3-Noodler or Z3.

Despite these achievements, there are limitations: we do not support scientific notation (exponential notation), the special values `NaN`/`Inf`, or infinite or periodic decimal expansions (except for special edge treatment). Therefore, in the future, the agenda is to add support for syntaxes such as `1.23e-5`, further optimization of length vector generation, and exploration of quasilinear techniques that would limit the number of disjunctions and contribute to scalability for very long strings.

In conclusion, it can be stated that accurate yet powerful support for string-to-real conversions is, in principle, achievable without major intervention in the solver core. Our solution brings new quality to the area of formal verification, where text and decimal numbers are commonly intertwined, and opens up space for practical verification of critical systems from web applications to embedded devices with strict numerical validation.

Bibliography

[1] Aho, A. V.; Lam, M. S.; Sethi, R. and Ullman, J. D. *Compilers: Principles, Techniques, and Tools*. 2nd ed. Pearson, 2006. ISBN 978-0321486813.

[2] Barrett, C. W.; Fontaine, P. and Tinelli, C. The SMT-LIB Standard: Version 2.6. In: *Proceedings of the 13th International Workshop on Satisfiability Modulo Theories (SMT 2016)*. 2016. Available at: http://smtlib.cs.uiowa.edu/papers/smtlib-v2.6-r2017-07-18.pdf.

[3] Chen, Y.-F.; Chocholatý, D.; Havlena, V.; Holík, L.; Lengál, O. et al. Solving String Constraints with Lengths by Stabilization. *Proceedings of the ACM on Programming Languages (OOPSLA)*. ACM, 2023, vol. 7, OOPSLA2, p. 1–31. Available at: https://dl.acm.org/doi/10.1145/3622872.

[4] Dantzig, G. B. Maximization of a Linear Function of Variables Subject to Linear Inequalities. *Activity Analysis of Production and Allocation*, 1947, p. 339–347. Introduces the simplex algorithm.

[5] Enderton, H. B. *A Mathematical Introduction to Logic*. 2nd ed. Academic Press, 2001. ISBN 978-0122384523.

[6] Ferrante, J. and Rackoff, C. W. A Decision Procedure for the First Order Theory of Real Addition with Order. *SIAM J. Comput.*, 1975, vol. 4, no. 1, p. 69–76.

[7] Fourier, J. and Motzkin, T. S. Elimination Theory for Linear Inequalities. *Translated Selections*, 1827. Combined historical sources on what is now called Fourier–Motzkin elimination.

[8] Havlena, V.; Holík, L.; Lengál, O. and Síč, J. Cooking String-Integer Conversions with Noodles. In: *27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024, vol. 305, p. 25:1–25:20. Leibniz International Proceedings in Informatics (LIPIcs). Available at: https://drops.dagstuhl.de/opus/volltexte/2024/20380/. Also available at https://github.com/VeriFIT/z3-noodler.

[9] Hopcroft, J. E.; Motwani, R. and Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation*. 3rd ed. Pearson, 2006. ISBN 978-0321455369.

[10] Junttila, T. *Notes on First-Order Logic*. 2020. Available at: https://users.aalto.fi/~tjunttil/2020-DP-AUT/notes-smt/fol.html. Lecture notes, accessed 2025-05-03.

[11] Kleene, S. C. Representation of Events in Nerve Nets and Finite Automata. *Automata Studies*. Princeton University Press, 1956, p. 3–41.

[12] 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: *CAV 2014*. 2014, vol. 8559, p. 205–221. LNCS.

[13] Moura, L. de and Bjørner, N. Z3: An Efficient SMT Solver. In: *TACAS 2008*. 2008, vol. 4963, p. 337–340. LNCS.

[14] Nelson, G. and Oppen, D. C. Simplification by Cooperating Decision Procedures. In: *POPL 1979*. 1979, p. 245–257.

[15] OWASP Foundation. *OWASP Top 10: 2021 – A03 Injection*. 2021. Available at: https://owasp.org/Top10/A03_2021-Injection/. Online; accessed 2025-05-03.

[16] Presburger, M. On the Completeness of a Certain System of Arithmetic. *Comptes Rendus du I Congrès de Math. des Pays Slaves*, 1930, p. 92–101. English translation in *History and Philosophy of Logic*, 2010.

[17] Rabin, M. O. and Scott, D. Finite Automata and Their Decision Problems. *IBM Journal of Research and Development*, 1959, vol. 3, no. 2, p. 114–125.

[18] Rungta, N.; Nelson, T.; Delaware, B.; Sullivan, D.; Bandi, P. et al. A Billion SMT Queries a Day. In: *CAV 2022*. 2022, vol. 13371, p. 3–18. LNCS.

[19] Sipser, M. *Introduction to the Theory of Computation*. 3rd ed. Cengage Learning, 2012. ISBN 978-1133187790.