TU WIEN Informatics

Solving String Equations

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Theodor Seiser, BSc
Matrikelnummer 01627752

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof.in Dr.in techn. Laura Kovács, MSc

Wien, 15. August 2025

Theodor Seiser	Laura Kovács

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Solving String Equations

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Theodor Seiser, BSc
Registration Number 01627752

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof.in Dr.in techn. Laura Kovács, MSc

Vienna, August 15, 2025

Theodor Seiser

Laura Kovács

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Theodor Seiser, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 15. August 2025

Theodor Seiser

Acknowledgements

I would like to thank all people that supported my writing this thesis. First and foremost, I am very grateful to my supervisor, Laura Kovács for her guidance and valuable feedback.

A special thank you goes to Clemens Eisenhofer for countless enlightening discussions on string solving, for generating ideas together and for proofreading this thesis. I would also like to thank Nikolaj Bjørner for skilled technical advice.

Finally, I would like to thank my partner and my family for their support.

Kurzfassung

Stringgleichungen, treten in vielen Bereichen der Softwareentwicklung und formalen Methoden auf. Sie spielen eine zentrale Rolle in Aufgaben wie Sicherheitsanalysen, automatisiertem Schließen und Softwareverifikation, bei denen die Erfüllbarkeit einer Bedingung wie $xy \simeq abc$ darüber entscheidet, ob ein bestimmtes Programmverhalten möglich ist. Eine Lösung für eine solche Bedingung ist eine Belegung der Variablen mit konkreten Zeichenketten, sodass die Gleichung erfüllt ist.

Diese Arbeit befasst sich eingehender mit der Nielsen-Transformation, einem verbreiteten Ansatz zum Lösen von Stringgleichungen. Die Nielsen-Transformation wendet systematisch Umformungsregeln an, um Gleichungen zu reduzieren, bis sie trivial erfüllbar oder unerfüllbar sind. Die dabei entstehenden Variablensubstitutionen können mit Hilfe eines Substitutionsgraphen organisiert und untersucht werden.

Ein weiterer bedeutender Ansatz ist Recompression, wo Gleichungen durch iteratives Ersetzen von wiederholten Mustern durch neue Symbole vereinfacht werden. Dadurch wird die Problemgröße reduziert, während die Erfüllbarkeit erhalten bleibt.

Ein dritter häufig verwendeter Ansatz nennt sich Stabilization. Solver modellieren dabei Variablendomänen als reguläre Sprachen und verwenden Automaten, um diese Domänen schrittweise zu verfeinern.

Wir erweitern die Menge der Regeln für die Nielsen-Transformations um eine parametrisierte Substitutionsregel, die unendliche Familien von Lösungen, die durch Zyklen im Substitutionsgraphen entstehen, kompakt darstellt. Dies ermöglicht es, Zyklen durch eine einzige Kante mit einer Potenzannotation zu ersetzen, wodurch Termination in Fällen möglich wird, in denen das ursprüngliche Verfahren divergieren würde.

Darüber definieren wir das Konzept der Signatur für eine Gleichung, die ein gemeinsames strukturelles Präfix verschiedener Gleichungen erfasst. Gleichungen mit derselben Signatur durchlaufen oft dieselbe Abfolge von Umformungsschritten, auch wenn sich ihre verbleibenden Teile unterscheiden. Durch das Gruppieren solcher Gleichungen lassen sich redundante Graphenerweiterungen vermeiden und der Suchraum verkleinern.

Durch die Signatur wird das Problem in Teilprobleme geteilt und es ergeben sich Teillösungen. Zur kompakten Darstellung dieser Teillösungen verwenden wir EDT0L-Sprachen, die einen formalen, sprachtheoretischen Rahmen bieten, um potenziell unendliche Mengen von Lösungen strukturiert zu erfassen.

Abstract

String equations arise in many areas of software engineering and formal methods. They play a central role in tasks such as security analysis, automated reasoning, and software verification, where the satisfiability of a constraint like $xy \simeq abc$ determines whether a program behavior is possible. A solution to such a constraint is an assignment of concrete strings to variables that satisfies the equation.

This thesis takes a closer look at Nielsen transformation, a common approach to solving string equations. Nielsen transformation applies systematic rewriting rules to reduce equations until they become trivially satisfiable or unsatisfiable. The rules may lead to variable substitutions, which can be organized and explored using a substitution graph.

Another prominent approach is recompression, where equations are simplified by iteratively replacing repeated patterns with fresh symbols, reducing problem size while preserving satisfiability.

A third approach, often used by solvers, is called stabilization. Variable domains are represented as regular languages and automata are used to iteratively refine these domains.

We extend the standard Nielsen transformation rule set with a parameterized substitution rule that compactly represents infinite families of solutions arising from cycles in the substitution graph. This allows cycles to be replaced by a single edge annotated with a power expression and enabling termination in cases where the unmodified procedure would diverge.

We define a signature for an equation, representing a common structural prefix of different equations. Equations sharing the same signature evolve through the same repeating sequence of rewriting steps, even if their trailing terms differ. By grouping such equations, we avoid redundant graph expansion in certain cases and reduce the size of the search space.

By using the signature the problem is divided into subproblems that lead to partial solutions. To represent these partial solutions compactly, we use EDT0L languages, which provide a formal language-theoretic framework to capture potentially infinite sets of solutions in a structured way.

Contents

CHAPTER 1

Introduction

String data types are an omnipresent utility in software programs [HRS19]. However, they impose certain challenges in software verification. The correctness of a program might depend on the satisfiability of a string equation or another string constraint, such as length constraints or membership in a regular language. For example when checking passwords in distributed systems. A verification tool might have to check an assertion like `assert(concat(x,y)="abc");` where x and y are program variables of data type string. The variable assignments that satisfy this assertion are $x = "" \wedge y = "abc", x = "a" \wedge y = "bc", x = "ab" \wedge y = "c", x = "abc" \wedge y = ""$. From a satisfiability standpoint, these variable assignments are satisfying models for the string equation $xy = abc$. Note that juxtaposition is concatenation and we always use a, b and c for character constants and x, y and z for string variables. We omit the double quotes around character constants for simplicity.

The previous example shows that for solving these equations an alignment among string variables is needed where the left-hand side matches the right-hand side. This assignment problem becomes much harder if there are variables on both sides of the equation and if the same variable occurs more than once.

Besides formal software verification [HRS19, AAC+14, CHL+19] string solving is used for various applications, including security analysis [EMS07, ESM+23, TCJ14, WS07], and automated reasoning [Ama23, Hag19]

1.1 State of the art

The SMT solver Z3 [dMB08a] has a string solver implementation that uses conflict learning to gradually refine models. Other solvers like S3 [TCJ14] and Z3Str4 [MBK+21] are built on top of Z3 to leverage the advantages of SMP theory solvers and support multiple theories.

Another SMT solver, cvc5 [BBB+22], has integrated reasoning over the theory of strings together with length and membership in regular languages into its DPLL(T) framework [LRT+14].

NOODLER [BCC+23] assigns regular membership constraints on variables and reasons about the corresponding automaton to refine these constraints.

OSTRICH [CHL+19] has a similar approach using regular automatons. In its core it uses backward propagation to repeatedly create pre-images of regular expression constraints.

1.2 Thesis Objective and Outline

The objective of this thesis is to provide an in-depth overview of approaches to solving string equations, compared to our own contributions. In particular we exploit and expand Nielsen transformation and power term manipulation in our work.

In chapter 3, we begin by reviewing three prominent approaches to string solving—namely, Nielsen Transformation, recompression, and automata-based methods. This includes a formal description of each method's operational principles, supported by illustrative examples.

These techniques are then extended by insights gained during the development of a new string solver within the Z3 framework, as documented in our publication [ESBK25]. The thesis presents techniques that proved effective in practice, as well as those that could not be successfully leveraged.

1.3 Contributions

The main contributions of this thesis are:

- Extending Nielsen transformation with power substitution (section 4.2)
  A new rule is added that introduces power terms of the form w^k to avoid repeatedly applying the same rule in a loop and can improve termination behavior.
- Handling of power terms in equations (section 4.2)
  The formalism is extended to deal with those newly introduced power terms, allowing direct manipulation of repeated patterns and enabling constraint-based branching on integer exponents.
- Using signatures to compress substitution graphs (section 5.2)
  A method for grouping equations with the same *signature* (structural prefix) is proposed, reducing infinite graph expansions in some non-terminating cases.
- Representation of variable constraints in EDT0L grammars. (section 5.3)
  Partial solutions that arise from substitution graphs using signatures are expressed as a language generated by a EDT0L (Extended Deterministic Table 0-interaction Lindenmayer) grammar [DJK16].

CHAPTER 2

Preliminaries

2.1 Basic Definitions

Let Σ be a finite and non-empty alphabet. The elements of Σ are called *characters* and are denoted by the lowercase letters a, b and c. We assume all character constants are pairwise distinct, i.e., $a \neq b$, $a \neq c$, and so on.

Let V be a finite set of *string variables*, whose elements are denoted by the lowercase letters x, y and z. A *token* is either a character $a \in \Sigma$ or a string variable $x \in V$.

The concatenation of two terms is denoted by the operator $\cdot$. Concatenation is associative but not commutative. The *empty string* is denoted by ε, which acts as the neutral element for concatenation:

$$u \cdot \varepsilon = \varepsilon \cdot u = u.$$

A *string term* is defined inductively as:

- a token,
- the empty string ε,
- or the concatenation of two string terms.

By associativity, a term can be viewed as a sequence of tokens. We write

$$u = t_0 \cdot t_1 \cdot \ldots \cdot t_{k-1}$$

and refer to this syntactic token sequence as $\langle u \rangle$. The *syntactic length* of u, written $|\langle u \rangle|$, is the number of tokens in u.

When clear from context, we omit the angle brackets and write concatenation as simple juxtaposition: $u = t_0 t_1 \ldots t_{k-1}$. Concatenations with ε are omitted. Only when $|\langle u \rangle| = 0$, we write $u = \varepsilon$.

Power expressions with integer exponents are used to denote consecutive occurrences of the same term and are defined inductively as $u^0 = \varepsilon$ and $u^{k+1} = u^k \cdot u$.

A *string equation* is an expression $u \simeq v$, where u is the *left-hand side (LHS)* and v is the *right-hand side (RHS)*, and u, v are string terms. Two different symbols are used for equality. The symbol $=$ is used for equality on a meta level as well as for definitions. The symbol $\simeq$ is used for string equations.

A *substitution* σ is a mapping from variables to string terms. We write $\sigma = [x/u]$ to denote a substitution where $\sigma(x) = u$, and $\sigma(y) = y$ for all other variables $y \neq x$. Substitutions apply homomorphically to terms and equations:

$$\begin{aligned} x[\sigma] &= \sigma(x) \quad \text{for variables,} \\ a[\sigma] &= a \quad \text{for characters,} \\ (u \cdot v)[\sigma] &= u[\sigma] \cdot v[\sigma], \\ (u \simeq v)[\sigma] &= u[\sigma] \simeq v[\sigma]. \end{aligned}$$

A substitution $\sigma_1 \circ \sigma_2$ denotes function composition, where

$$(\sigma_1 \circ \sigma_2)(x) = \sigma_1(x)[\sigma_2] \quad \text{for all } x \in V.$$

A substitution σ is a *model* or *satisfying interpretation* of the equation $u \simeq v$ if the evaluated terms are syntactically equal: $\langle u[\sigma] \rangle = \langle v[\sigma] \rangle$. A set of equations is called *satisfiable* if there exists a substitution that is a model for all equations in the set. Otherwise, the set is *unsatisfiable*.

We define the constants $\top$ and $\bot$ to represent trivially true and false equations, respectively.

A string term is called ground if it contains no variables.

In addition to syntactic length, we define the *string length* of a term, written $|u|$, as the number of characters in the fully evaluated ground string. That is, unlike syntactic length, which counts tokens (variables and constants), string length counts actual characters in the interpreted string. The string length of variables or string expressions containing variables might be unknown during the solving process.

A graph is defined as a tuple (N, E) where $n \in N$ are the vertices and $(n_0, e, n_1) \in E$ are the edges.

2.2 Core Principles of String Solving

The fundamental result by Gennadiy Semenovich Makanin [Mak77] established that satisfiability of existential quantified word equations is decidable. Subsequent work refined this result, showing that the problem lies in PSPACE [Pla99], and further algorithmic improvements have made the problem more practical in constrained settings [Gut98].

The set of satisfying models may be infinite, and solvers often aim describe the solution set, or at least a subset, symbolically. This can be done using variables in a solution representation which may be unrestricted or restricted by additional constraints. Solutions can also be represented using regular expressions or even more involved languages using for example an EDT0L grammar [DJK16].

As discussed by Day [Day22], the general problem of string solving can be viewed through the interaction of two dimensions:

- Syntactic structure of the equation including the number of variables, repeated occurrences, and the ordering of constants and variables.
- Additional constraints such as string lengths (e.g., $|x| = 3$), regular membership (e.g., $x \in (ab)^*$), or string functions (e.g., `replaceAll`, `reverse`).

Length constraints naturally arise from string equations as an equations $u \simeq v$ implies that $|u| = |v|$. On the other hand solvers often allow user input to include length constraints.

In the absence of additional constraints, word equations are solved purely by structural reasoning. In practice, however, string solvers combine structural unification with auxiliary theories (e.g., arithmetic or regular languages) to guide and prune the search space [DKM+20].

Several major solving strategies exploit different aspects of the problem structure. Three main approaches are presented in chapter 3:

- *Nielsen Transformation* applies rule-based substitutions based on the first tokens on both sides [Nie17].
- *recompression* simplifies equations through compression of character pairs and blocks [Jeż16].
- *Automata-based solvers* [BCC+23, CHL+19]) use regular constraints to refine variable domains and reason about satisfiability through language inclusion.

A key observation in [Day22] is that many real-world string constraints encountered in verification problems fall into decidable and tractable fragments. Examples include the *quadratic fragment*, where each variable occurs at most twice, and fragments that disallow mutual recursion or looping in substitution graphs which will be described in subsection 3.1.3.

Modern string solvers are often included in Satisfiability Modulo Theories (SMT) engines. These solver frameworks combine different theories like integer arithmetic, bit vectors, strings and quantifiers via DPLL(T) [BBB+22, dMB08b, Rüm08]. Integer arithmetic is especially interesting in a string solving context dealing with length constraints.

CHAPTER 3

Existing Approaches to String Solving

3.1 Nielsen Transformation

3.1.1 Standard Rules

Nielsen Transformation is a set of rules to transform the equation [Nie17]. There are various different formulations of these rules, and we present our own extended set in chapter 4. The rules here can be seen as a kind of common ground in literature and are necessary and sufficient to cover all possible cases. The first six rules simplify the equation in a straightforward way; therefore, they are called *simplifying rules*.

$$\varepsilon \simeq \varepsilon \rightsquigarrow \top \tag{3.1}$$
$$au \simeq \varepsilon \rightsquigarrow \bot \tag{3.2}$$
$$au \simeq bv \rightsquigarrow \bot \tag{3.3}$$
$$au \simeq av \rightsquigarrow u \simeq v \tag{3.4}$$
$$xu \simeq xv \rightsquigarrow u \simeq v \tag{3.5}$$
$$xu \simeq \varepsilon \rightsquigarrow u[x/\varepsilon] \simeq \varepsilon \tag{3.6}$$

Two cases are not yet covered by these rules. The first is $xu \simeq av$. For the assignment of the variable x in any satisfying model, there are two possibilities. Either x is empty and we can replace every x with ε or x starts with a in which case x can be replaced by ax'. Since x' is a fresh variable and x does not appear after its replacement, we can reuse the symbol x. So we replace x with either ax or with ε. We get a new rule.

$$xu \simeq av \rightsquigarrow (xu \simeq av)[x/ax] \vee (xu \simeq av)[x/\varepsilon] \tag{3.7}$$

The last case to consider is $xu \simeq yv$. One or both of the variables might be empty. Otherwise, either x is longer or equal to y and therefore starts with y or y is longer or equal to x and starts with x. Similarly to the previous rule, if x starts with y we replace x by yx and vice versa.

$$xu \simeq yv \rightsquigarrow (xu \simeq yv)[x/yx] \vee (xu \simeq yv)[y/xy] \vee (xu \simeq yv)[x/\varepsilon] \vee (xu \simeq yv)[y/\varepsilon] \quad (3.8)$$

After the substitution x/ax from rule (3.7) we can always apply rule (3.4). And after x/yx or y/xy from rule (3.8) we can apply rule (3.5). In later examples, we will implicitly apply these simplification rules immediately after the splitting rule.

For example for the equation $x = ay$ the rule (3.7) will give $\varepsilon \simeq ay \vee ax \simeq ay$. This can immediately be simplified to $x \simeq y$.

For rules (3.7) and (3.8) the algorithm has to split on which branch to take, and if the branch fails it has to backtrack and try another branch, we call them *splitting rules*. They are deliberately formulated in a way that for every case there is exactly one substitution that is applied to the whole equation and nothing else is done. This allows us to say for rule (3.6) that the equation $xu \simeq av$ suggests the substitutions x/ax and x/ε and for rule (3.7) that $xu \simeq yv$ suggests $x/yx, y/xy, x/\varepsilon$ and y/ε.

Note that the rules (3.2), (3.6) and (3.7) have symmetric versions where the left and right hand side are swapped.

To illustrate the concept, consider the example $xx \simeq aa$. We apply rule (3.7) which, when we apply it to the equation, gives us $xx \simeq aa \rightsquigarrow (xx \simeq aa)[x/ax] \vee (xx \simeq aa)[x/\varepsilon]$. We perform the replacements and get $xx \simeq aa \rightsquigarrow axax \simeq aa \vee \varepsilon = aa$. For the equation $\varepsilon \simeq aa$ rule (3.2) is applicable, which immediately gives us $\bot$. Only the equation $axax \simeq aa$ remains. Rule (3.4) simplifies it to $xax \simeq a$. Applying this to rule (3.7) gives us $xax \simeq a \rightsquigarrow axaax \simeq a \vee a \simeq a$. Note that the term u from the general rule is here ε and omitted. We can immediately see that we have the trivially satisfiable equation $a \simeq a$. Nielsen Transformation will simplify this to $\top$ after applying rule (3.4) and then rule (3.1).

3.1.2 Reusing Variables

In rule (3.7) we have the case that we replace x by ax. We call this fixing the prefix of x because in every solution that follows after this replacement, the interpretation of the original x starts with an a. We reused the variable x instead of introducing a fresh variable x'. This might seem haphazard and prone to confusion, but it has a good algorithmic justification. The same holds for rule (3.8) when we replace x by yx as well as the symmetric case.

We take a look at another simple example. Consider the equation $xb \simeq ax$. This is obviously unsatisfiable no matter what x is. There is one more a on the left hand side than on the right. If we used Nielsen Transformation to solve this we would apply rule

(3.7) and get two cases. The second case is easier so we start here. We get $b \simeq a$ which is unsatisfiable so we backtrack to the case distinction. After applying the first case we again get the equation $xb \simeq ax$. We have the same equation as at the start, so we can repeatedly apply the first case of rule (3.7). However, whenever we apply the second case we get $b \simeq a$, so we will never find a solution.

Whenever we detect that we have the same equation as before we can stop, to avoid running in a loop. If we had introduced fresh variables, the new equation in our example would have looked like $x'b \simeq ax'$. We cannot immediately see that it is the same as before. We would have to check if the equations are equivalent modulo variable renaming, which becomes increasingly difficult with every new variable we introduce.

3.1.3 Substitution Graph

Proofs using rule systems are typically illustrated using proof trees. However, if some substitutions lead to a loop (i.e. cyclic rewriting), this should be represented in the graph as a cycle. A more general directed graph, called substitution graph, is used that allows cycles. The nodes of the substitution graph are labeled by equations, and the edges are labeled by substitutions. An edge is defined as a triple (n_1, σ, n_2) where n_1 is the outgoing node, σ a substitution and n_2 the ingoing node. This can be interpreted as applying the substitution σ to the equation at n_1 gives us the equation at n_2.

The substitution graph is constructed as follows. We start with a root[1] node and label it with the input equation. As long as simplifying rules can be applied, we apply these rules directly to the label of the node. As long as we have an unprocessed node n_0 where a splitting rule is applicable, we create a new node n_i and an outgoing edge (n_0, σ_i, n_i) for every substitution σ_i that is suggested by the equation. We immediately apply all possible simplifying rules to the new n_i. If n_i is equivalent to an existing node n_j we remove n_i and the edge to it and instead create an edge (n_0, σ_i, n_j). The node n_0 is then marked as processed.

A node can be labeled with one of three types of equations. It can be an equation where a splitting rule is applicable. The node will have outgoing edges after it is processed and is called an inner node. It can be $\bot$; the node is then called a conflict node. Or it can be $\top$ and the node is a witness node. Any other equation could be further simplified. Both witness nodes and conflict nodes are terminal nodes.

If there is a witness reachable from the root node, we have shown that the equation is satisfiable. If all inner nodes are processed and all terminal nodes are conflict nodes, we have shown unsatisfiability. This is, in general, not a terminating decision procedure, as we could create infinitely many nodes. In our examples, the labels of the terminal nodes are not simplified to $\top$ or $\bot$. Instead, witness nodes are marked with a star * and conflict nodes are marked with a dagger †. This is to better show why the equation is satisfiable or unsatisfiable.

[1] Although the graph is not a tree, it still makes a lot of sense to use the term root. The node is uniquely defined with the input equation and all other nodes can be reached from it.

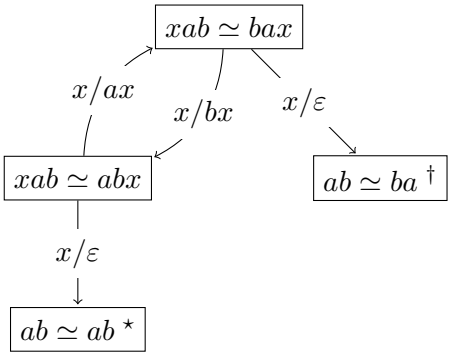


Figure 3.1: Substitution graph for $xab \simeq bax$

Figure 3.1 shows an example of a substitution graph for the equation $xab \simeq bax$. There are two nodes where rule (3.7) is applicable, so they have two outgoing edges. Applying substitution x/ax to $xab \simeq abx$ gives, after simplification, the equation $xab \simeq bax$. There already is such a node, the root node, so the edge points to this existing node. After adding this edge the graph contains a cycle. We also have one conflict node and one witness node showing that the equation is satisfiable. We can also determine models from the substitution graph. This is done by chaining all substitutions on a path from the root node to a node with label $\top$. For the shortest path from $xab \simeq bax$ to $\top$ we get $\sigma = [x/bx] \circ [x/\varepsilon] = [x/b]$. So one possible solution is $\sigma(x) = b$. Using different paths, by going a certain number of iterations in the cycle, we can get different models. In fact, the substitution graph gives us all satisfying models [LM21].

3.1.4 Generalization to Multiple Equations

The substitution graph with Nielsen Transformation can be easily generalized to accept multiple equations as input. Every node is now labeled by a set of equations **EQ**; the root node is labeled by the set of input equations. Simplification rules are applied to every equation in **EQ** until no simplification rule is applicable at any equation. The order in which they are applied is irrelevant. Whenever rule (3.6) it is applicable to one equation, the substitution x/ε has to be applied on all equations.

A node is a witness if all equations are $\top$. If at least one equation is $\bot$, the node is a conflict node. Otherwise, the node is an inner node. An inner node n_0 might have equations that are $\top$. For every other equation, there is one splitting rule that is applicable. For every substitution σ_i that is suggested by any of these equations a new node and an outgoing edge (n_0, σ_i, n_i) are created. The substitution σ_i is applied to all equations. Similar to before, n_i will be replaced by an existing node n_j if the labels – in this case, the set of equations – are equal.

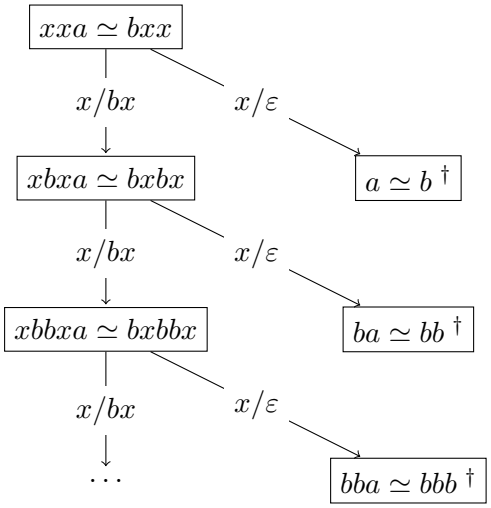


Figure 3.2: Substitution graph for $xxa \simeq bxx$

3.1.5 Termination on the Quadratic Fragment

In subsection 3.1.3 we have seen that a substitution graph can tell us if an equation is satisfiable and, in case it is, also gives us all models. However the algorithm for constructing the substitution graph is not always terminating. Consider the equation $xxa = bxx$ in Figure 3.2. After applying rule (3.7) we have a new node with the equation $xbxa = bxbx$ and after another application of rule (3.7) a node with $xbbxa = bxbbx$. We can start to see a pattern. We generate longer and longer equations but the first token on both sides is always the same so we always apply the same rule. The branches where we apply the substitution x/ε will always be labeled with $\bot$ since there is no variable and there is an a on the left hand side and not on the right hand side.

However, we can guarantee termination if we restrict the input to the so-called quadratic fragment. A set of input equations is in the quadratic fragment if every variable appears at most twice in total over all equations from the set.

To prove this, we first state the following lemma.

Lemma 3.1.1 *Let* **EQ** *be a quadratic set of equations. If* σ *is a substitution suggested by one of the splitting rules* (3.7)-(3.8) *for an equation* $eq \in$ **EQ**, *then there exists a simplification rule* π_i (3.1)-(3.6) *such that* $\pi_i(\mathbf{EQ}[\sigma])$ *is still quadratic and contains no more tokens than* **EQ**.

There are three types of substitutions that can be suggested by a splitting rule:

- The substitution x/ε removes all occurrences of the variable x from every equation in **EQ**. It introduces no additional tokens and strictly decreases the token count.

No new variable appearance is introduced, so the set of equations $\mathbf{EQ}[\sigma]$ is still quadratic.

- The substitution x/ax introduces at most two additional tokens (two occurrences of the constant a), since x occurs at most twice in $\mathbf{EQ}$ due to the quadraticity assumption. This substitution is suggested by an equation of the form $xu \simeq av$. Therefore, such an equation must exist in $\mathbf{EQ}$, and in $\mathbf{EQ}[\sigma]$ we find the transformed equation $axu[x/ax] \simeq av[x/ax]$.

  This equation can be simplified using rule (3.4), yielding $xu[x/ax] \simeq v[x/ax]$. The simplification removes two occurrences of a. As a result, the overall number of tokens is either reduced or left unchanged, and the number of variable occurrences remains the same. Thus, $\pi_i(\mathbf{EQ}[\sigma])$ is still quadratic.

- The substitution x/yx introduces at most two additional tokens y, again because x occurs at most twice. This substitution is suggested by an equation of the form $xu \simeq yv$, handled by rule (3.8). In $\mathbf{EQ}[\sigma]$, the substitution results in an equation of the form $yxu[x/yx] \simeq yv[x/yx]$, which can then be simplified using rule (3.5) to eliminate two occurrences of y.

  As in the previous case, the simplification cancels out the additional tokens introduced by the substitution, resulting in no net increase in token count. The variable count remains bounded, and the system remains quadratic.

In all three cases, the simplification reduces or maintains the number of tokens and preserves quadraticity.

From this lemma follows via induction over the construction of the graph that if a root node is labeled by a quadratic set of equations all other nodes are quadratic and the number of tokens in any node is smaller than or equal to the number of tokens in the root. Since the number of tokens is limited, there are only finitely many possible different equations. Also, the number of equations per set is limited since we only replace equations and do not introduce new ones. With finitely many equations to choose from and a limited set size, there are only finitely many different sets of equations. Since the nodes of the substitution graph are labeled by unique equation sets the graph is finite. This means eventually every node will be processed and the algorithm terminates.

3.2 Recompression

Recompression is a powerful technique for solving word equations, introduced by Artur Jeż in [Jeż16]. The core idea is to simplify the structure of the equation through systematic compression of repeated patterns in the words. The technique is based on two operations:

- *Pair compression*: All occurrences of a pair of distinct letters $ab \in \Sigma^2$ are replaced by a fresh symbol $c \notin \Sigma$.

- *Block compression*: Maximal blocks a^ℓ of the same letter $a \in \Sigma$ are replaced by a fresh symbol a_ℓ.

Given a word equation $u \simeq v$, a *solution* is a morphism $S\colon \mathcal{V} \to \Sigma^*$ such that both $S(u)$ and $S(v)$ are ground terms and $S(u) = S(v)$.

Both pair and block compression are applied simultaneously to all eligible substrings in $S(u)$ and $S(v)$. Recompression distinguishes three types of occurrences of substrings $w \in \Sigma^+$ in the expression $u \simeq v$:

- *Explicit* if w occurs directly as a substring of u or v (i.e., outside of variables).
- *Implicit* if w appears as a substring entirely within the image $S(x)$ of some variable x.
- *Crossing* if w straddles a boundary between variables and/or constants, e.g., one letter comes from $S(x)$ and the other from elsewhere in the equation.

A key challenge in recompression is dealing with crossing appearances. For example, a crossing occurrence of a pair ab may arise if:

- A variable x ends with a, and the next symbol in the equation is b, or
- A variable x starts with b, and the preceding symbol is a, or
- Two adjacent variables x and y are such that $S(x)$ ends with a and $S(y)$ starts with b, and xy appears in u.

Crossing appearances cannot be directly replaced, so the algorithm modifies the structure of u to make the critical characters explicit. If x ends with a, and the following symbol is b, then every occurrence of x in the equation is replaced by $x' = xa$, and the trailing a is removed from $S(x)$. If x starts with b, and the preceding symbol is a, then x is replaced by $x' = bx$, and the leading b is removed from $S(x)$.

This process is referred to as *left-popping* and *right-popping*, respectively, depending on whether the letter is moved out of the start or end of the variable. Once all crossing appearances of a pair ab are converted to explicit ones, they can be safely replaced during pair compression.

Block compression operates similarly. For a maximal block a^ℓ, if part of the block is hidden inside a variable x, left-popping and right-popping are applied repeatedly to extract all consecutive a's from the boundary of $S(x)$ until a different symbol is encountered. This makes the block explicit and it can now be compressed.

The recompression procedure repeatedly applies pair and block compression steps, transforming the equation into a simpler form. Additionally, if the variable x has the solution $S(x) = \varepsilon$, x is removed from the equation.

The algorithm continues until the equation becomes trivial. That is when both sides reduce to the same constant string.

Importantly, the recompression process operates under the assumption of a *length-minimal solution*. However, since the actual solution is not known in advance, decisions regarding popping letters and removing empty variables are inherently *non-deterministic*. The algorithm explores different branching choices in a fair way, ensuring that a correct solution path will eventually be discovered, if one exists.

3.3 Automata-Based Methods

String equations as well as many string constraints can be expressed using regular languages. Many solvers use automata to represent and reason over regular languages. One approach, using automata, is called stabilization and is used in state of the art solvers like NOODLER [BCC+23, CCH+24]. This method combines word equations with regular language constraints, leveraging automata to represent and refine possible solutions.

In this approach, each variable $x \in \mathcal{V}$ is initially assigned a regular language, which represents the set of all possible strings that x may take. If a variable is unrestricted by any additional constraints, it is initially assigned the language Σ^*, where Σ is the alphabet. The language of concatenated terms is defined via language concatenation:

$$L_1 \cdot L_2 = \{uv \mid u \in L_1, v \in L_2\}.$$

An assignment of languages to variables is *stable* if for every equation $u \simeq v$, the languages of the two sides coincide:

$$Lang(u) = Lang(v),$$

which means

$$Lang(u) \subseteq Lang(v) \quad \text{and} \quad Lang(v) \subseteq Lang(u).$$

When the inclusion $Lang(u) \subseteq Lang(v)$ is violated for some equation $u \simeq v$, the language assignment is refined to restore stability. The refinement is performed by intersecting the languages $Lang(u)$ and $Lang(v)$ using the product construction on their corresponding automata.

A key feature in this process is how variable boundaries within u are represented. In the automaton recognizing the language that will be refined $Lang(u)$, edges labeled with ε mark the borders between occurrences of variables. These ε-edges are preserved in the product automaton to maintain the separation of variable segments.

Using these ε-edges, the product automaton is split into subsequences, each corresponding to a single occurrence of a variable within u. When a variable appears multiple times, the languages corresponding to its occurrences are intersected to enforce consistency. Because multiple ways to split the automaton may exist, the procedure may need to branch over the different possible decompositions.

If the resulting language assigned to a variable becomes empty in any branch, that branch is discarded. Otherwise, the language assignment is refined by updating the variable's language accordingly.

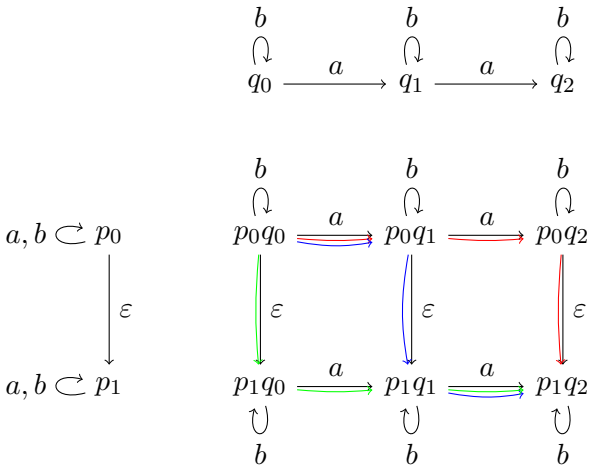


Figure 3.3: Product automaton illustrating stabilization

Consider the example with the equation

$$x \simeq yy$$

and the regular constraint

$$x \in b^*ab^*ab^*,$$

where $\Sigma = \{a, b\}$.

Initially, there are no constraints on y, so $y \in \Sigma^*$. Thus,

$$Lang(x) = b^*ab^*ab^*, \quad Lang(yy) = (a, b)^*.$$

Applying stabilization requires checking the subset relations:

$$Lang(x) \subseteq Lang(yy) \quad \text{and} \quad Lang(yy) \subseteq Lang(x).$$

The first inclusion is satisfied since language concatenation does not preserve the information that the both occurrences of y have to be equal. The second inclusion is not satisfied, so we refine $Lang(yy)$ by intersecting it with $Lang(x)$.

To do this, we construct the product automaton of $Lang(x)$ and $Lang(yy)$ as shown in Figure 3.3. In [BCC+23] similar illustrations are used to demonstrate different examples. Since yy is a concatenation of two occurrences of y, the ε-edges that mark the boundary between the two occurrences of y are preserved in the product automaton.

The automaton shows three distinct ε-edge paths from the initial to the final states, highlighted in red, blue, and green. Each path represents a possible way to split the language between the two y variables.

- Green path: The first y corresponds to language b^*, and the second y to $b^*ab^*ab^*$. Intersecting these gives
$$y \in b^* \cap b^*ab^*ab^* = \emptyset,$$
so this branch is discarded.
- Red path: Symmetrical to the green path, this yields
$$y \in b^*ab^*ab^* \cap b^* = \emptyset,$$
which is also discarded.
- Blue path: The first y corresponds to b^*ab^*, and the second y also corresponds to b^*ab^*. Their intersection is
$$y \in b^*ab^*,$$
which is non-empty and therefore constitutes a valid refinement.

Thus, the stabilization process refines y to the language b^*ab^*.

CHAPTER 4

Extending Nielsen Transformation

4.1 Power Substitutions

In subsection 3.1.3 it was shown that the substitution graph can contain cycles. One class of equations where such cycles occur are equations of the form $wxu \simeq xv$, with $w \in \Sigma^+$. To guarantee that we can introduce a cycle and the algorithm will not diverge u and v must not contain the variable x. With this restriction, we have the nice property that $u[x/w_i] = u$ and $v[x/w_i] = v$. Later, we will relax this restriction. This particular class is amenable to a specialized additional rule that complements the existing ones. By applying this rule, the introduction of cycles in these cases can be avoided while preserving the set of solutions. This leads to a more compact representation of the solution space and improves overall performance.

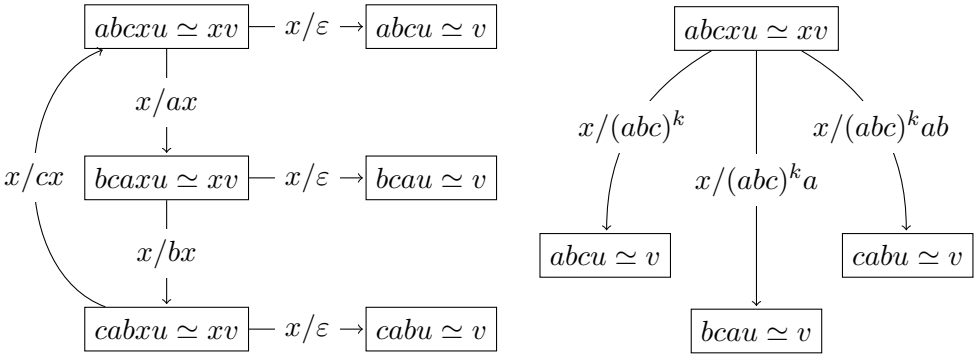
(a) Explicit path with loop. (b) Compact representation using σ_k.

Figure 4.1: Two representations of the substitution graph for the equation $axu \simeq xv$.

To motivate the additional rule, consider a special case of the class of equations of the form $wxu \simeq xv$, specifically when $w = a$, as illustrated in Figure 4.1a. In this case, the equation $axu \simeq xv$ is not a terminal node in the substitution graph, and it has a unique successor node, namely $au \simeq v$. For every solution of $axu \simeq xv$, the corresponding path in the substitution graph necessarily includes an edge labeled x/ε leading to $au \simeq v$. Prior to this edge, the path consists of $k \geq 0$ consecutive applications of the edge labeled

x/ax. In other words, each such path includes k transitions of the form x/ax, followed by a single transition of the form x/ε. These paths can be compactly represented by introducing a parameterized edge from $axu \simeq xv$ directly to $au \simeq v$, labeled with the substitution σ_k, where $\sigma_k = x/ax \circ x/ax \circ \ldots \circ x/ax \circ x/\varepsilon = x/a^k$ as shown in Figure 4.1b

This construction replaces the multiple-step paths with a single edge, simplifying the graph structure while preserving the semantics of the substitutions.



(a) Explicit path with loop. (b) Compact representation.

Figure 4.2: Two representations of the substitution graph for the equation $abcxu \simeq xv$.

To generalize this to equations where $|w| > 1$, we take a look at another example when $w = abc$. As shown in Figure 4.2 the circle consists of three nodes and there are three edges that lead out of the loop and to a node where x is eliminated. After k iterations in the loop, the original x starts with $(abc)^k$. Depending on which edge out of the loop the path takes, x can have an additional a or ab at the end.

Generally equations of the form $wxu \simeq xv$, with with $w \in \Sigma^+$, have $\sigma_k(x) = (w_1w_2)^k w_1$ where $w = w_1w_2$ and $|w_2| > 0$. Note that the case where $w_1 = w, |w_2| = 0$ is not considered since the same solution can be expressed with $w_2 = w, |w_1| = 0$ and incrementing k by one. Also note that $w = w_1w_2$ is not a substitution or an equation but a syntactic equivalence and all three symbols w, w_1 and w_2 are just representations of a sequence of characters. w_1 can be any proper prefix of w, including the empty prefix, so the algorithm will split into $|w|$ branches, where in every branch w_1 is a different prefix of w. This yields a new rule. Given $w \in \Sigma^+$

$$xu \simeq wxv \rightsquigarrow \bigvee_{w_1w_2=w, |w_2|>0} (xu \simeq wxv)[x/(w_1w_2)^k w_1] \tag{4.1}$$

After the substitution $x/(w_1w_2)^k w_1$ is applied the new equation is $(w_1w_2)^k w_1u \simeq w_1w_2(w_1w_2)^k w_1v$. This can be simplified to $u \simeq w_2w_1v$.

Using this new rule, there is no need to have the loop anymore. This means the restriction that u and v must not contain x can be relaxed. However when applying the rule the

substitution for x is now relevant for u and v and the equation will simplify to $u' \simeq w_2 w_1 v'$ where $u' = u[x/(w_1 w_2)^k w_1]$ and $v' = v[x/(w_1 w_2)^k w_1]$.

4.2 Handling Power Terms within Equations

An expression like w^k is not a formal string term but a set of string terms $\{\varepsilon, w, ww, \cdots\}$. Since there is a compact representation for that set, it is still reasonable to treat it as a term. To this end, extended string terms are introduced. A *power term* is a term of the form w^k where $w \in \Sigma^+$ is called the base and $k \geq 0$ is called the exponent. An *extended string term* can be a string term, a power term or a concatenation of two extended string terms. An extended string equation is an equation $u \simeq v \colon C$ where u and v are extended string terms and C is a set of integer constraints over the exponents appearing in u and v.

To generalize the Nielsen transformation that it works with extended string equations a few additional rules are required. All existing rules are kept. It is only noted that the integer constraints are left unchanged by these rules. Thus, all cases where neither side starts with a power term are already covered. So in this section, only the rules where one side starts with a power term are presented.

$$w^k u = w^k v \colon C \rightsquigarrow u = v \colon C \tag{4.2}$$

$$w^k u \simeq \varepsilon \colon C \rightsquigarrow u \simeq \varepsilon \colon C \cup \{k = 0\} \tag{4.3}$$

Rule (4.3) can be applied if one side is a power term w^k and the other side is empty. Since w is required to at least contain one element from Σ the only possibility for w^k to be empty is when $k = 0$.

The case where one side starts with a power term w^k and the other with a constant a requires a split. The exponent k could be zero, then the power term can be removed. If k is greater than zero, the term w^k is rewritten to ww^{k-1}. Rewriting where the first sequence w is pulled out of the power term is called unwinding. Immediately after this, either rule (3.3) or rule (3.4) can be applied.

$$w^k u \simeq av \colon C \rightsquigarrow u \simeq av \colon C \cup \{k = 0\} \vee ww^{k-1} u \simeq av \colon C \cup \{k > 0\} \tag{4.4}$$

If there are two power terms and they have the same base, it is possible to eliminate one power term. The rule splits which exponent is higher and then rewrites the higher one. Immediately after this rule (4.2) can be applied.

$$\begin{aligned} w^{k_1} u \simeq w^{k_2} v \colon C \rightsquigarrow\ & w^{k_2} w^{k_1 - k_2} u \simeq w^{k_2} v \colon C \cup \{k_1 \geq k_2\} \vee \\ & w^{k_1} u \simeq w^{k_1} w^{k_2 - k_1} v \colon C \cup \{k_2 \geq k_1\} \end{aligned} \tag{4.5}$$

If the two power terms have different bases one or both of them could have zero as exponent. Otherwise, the bases have to be compared. This is done via unwinding.

$$\begin{aligned} w_1^{k_1}u \simeq w_2^{k_2}v \colon C \rightsquigarrow\ & u \simeq v \colon C \cup \{k_1 = 0, k_2 = 0\} \vee \\ & u \simeq w_2^{k_2}v \colon C \cup \{k_1 = 0, k_2 > 0\} \vee \\ & w_1^{k_1}u \simeq v \colon C \cup \{k_1 > 0, k_2 = 0\} \vee \\ & w_1 w_1^{k_1-1}u \simeq w_2 w_2^{k_2-1}v \colon C \cup \{k_1 > 0, k_2 > 0\} \end{aligned} \tag{4.6}$$

If one side starts with a power term and the other with a variable x, there are two possibilities. If x is longer or equal to the power term w^{k_1} a substitution is applied to x to fix that x starts with w^{k_1}. If x is shorter than w^{k_1} the variable x is substituted by a prefix of w^{k_1} which can only be of the form $(w_1w_2)^{k_2}w_1$ where $k_2 < k_1$ and $w_1w_2 = w$. It is necessary to branch on what w_1 and w_2 are. To make the branches distinct w_2 is forced to be non-empty.

$$\begin{aligned} w^{k_1}u \simeq xv \colon C \rightsquigarrow\ & (w^{k_1}u \simeq xv)[x/w^{k_1}x] \colon C \vee \\ & \bigvee_{w_1w_2=w, |w_2|>0} (w^{k_2}w^{k_1-k_2}u \simeq xv)[x/w^{k_2}w_1] \colon C \cup \{k_1 - k_2 > 0\} \end{aligned} \tag{4.7}$$

There are equations where rule (4.4) would be applicable but not beneficial. For example equation $(ab)^{k_1} \simeq a(ba)^{k_2}$. The case where $k_1 = 0$ is a conflict. The other case gives $b(ab)^{k_1-1} \simeq (ba)^{k_2}$ after simplification. The same rule is applicable again and it gives $(ab)^{k_1-1} \simeq a(ba)^{k_2-1}$. The exponents changed, but no real progress was made if k_1 and k_2 are not bound. The power terms can be unwound infinitely. If the term $a(ba)^{k_2}$ is replaced by the equivalent term $(ab)^{k_2}a$, rule (4.5) can be applied. The resulting equations $(ab)^{k_1-k_2} \simeq a$ and $\varepsilon \simeq (ab)^{k_2-k_1}a$ will both quickly lead to a conflict.

The problem in this example was that there are different extended string term representations that are equivalent, and depending on the representation, a different rule is applicable. Generally, the expressions should be in a form that rule (4.5) is preferred over unwinding. Therefore equivalence replacements are introduced that bring the equation is such a form.

$$w_1(w_2w_1)^k \rightsquigarrow (w_1w_2)^k w_1 \tag{4.8}$$

$$ww^k \rightsquigarrow w^k w \tag{4.9}$$

$$w^1 \rightsquigarrow w \tag{4.10}$$

Simplification rules are still prioritized so these replacements are done after no simplification is applicable but before any other rule. An equivalence replacement creates a new node with an edge labeled *rewrite*. The new node is processed like any other node, starting with the application of simplification rules.

To integrate the new rules into the substitution graph, it is necessary to categorize the new rules. Rule (4.1) and rule (4.7) are *splitting rules* and rule (4.2) as well as rule (4.3) are *simplification rules*. The rules (4.4) and (4.5) are similar to splitting rules however there are no substitutions. Instead each brach is characterized by integer constraints that are added to the set. Consequently we say they suggest integer constraints. Rules (4.4) and (4.5) are thus called *constraint splitting rules* and the edges on the graph are labeled with the suggested integer constraints.

The nodes now contain extended string terms thus their label also shows the constraints. If the constraints in the node are not consistent the node is a conflict node.

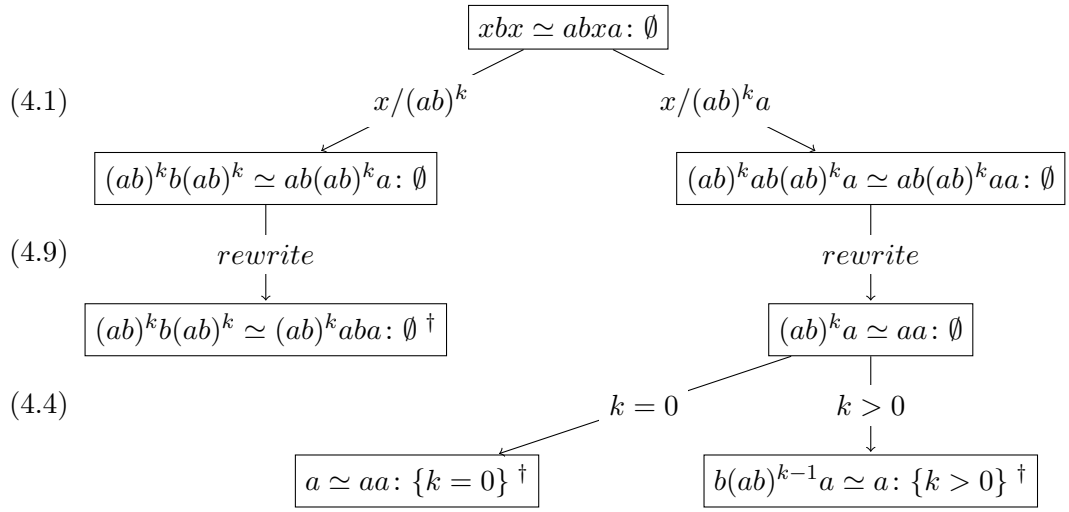


Figure 4.3: Example substitution graph with power introduction.

The example $xbx \simeq abxa$ in Figure 4.3 is unsatisfiable. It is not quadratic so Nielsen transformation is not guaranteed to terminate. And in fact rules (3.1)-(3.8) will only terminate on this example if rules (4.1)-(4.10) are used. The power is already introduced in the first step. The base is ab and there are two possibilities for the prefix w_1 namely ε or a. So there is a two way split. There is no simplification possible so both branches are rewritten using an equivalence replacement such that $(ab)^k$ moves to the start of the right hand side. The left branch simplifies to $\bot$ since after the first $(ab)^k$ the characters b and a clash. The right branch simplifies to $(ab)^k a \simeq aa$. Since no further simplification rules and no equivalence replacement is applicable, an unwinding rule has to be applied. Both the branch where $k = 0$ and the branch, where the first sequence is unwound from the power term, simplify to $\bot$. All inner nodes are processed and all terminal nodes are conflict nodes. Thus unsatisfiability was shown.

CHAPTER 5

Beyond Nielsen Transformation

5.1 Beyond Simple Loops

There are equations where the substitution graph has loops that can not be replaced by a power introduction rule from section section 4.1. Take a look at the following example $xabyu \simeq ybaxv$ where u and v do not contain the variables x or y. A part of the substitution graph for this equation is shown in Figure 5.1.

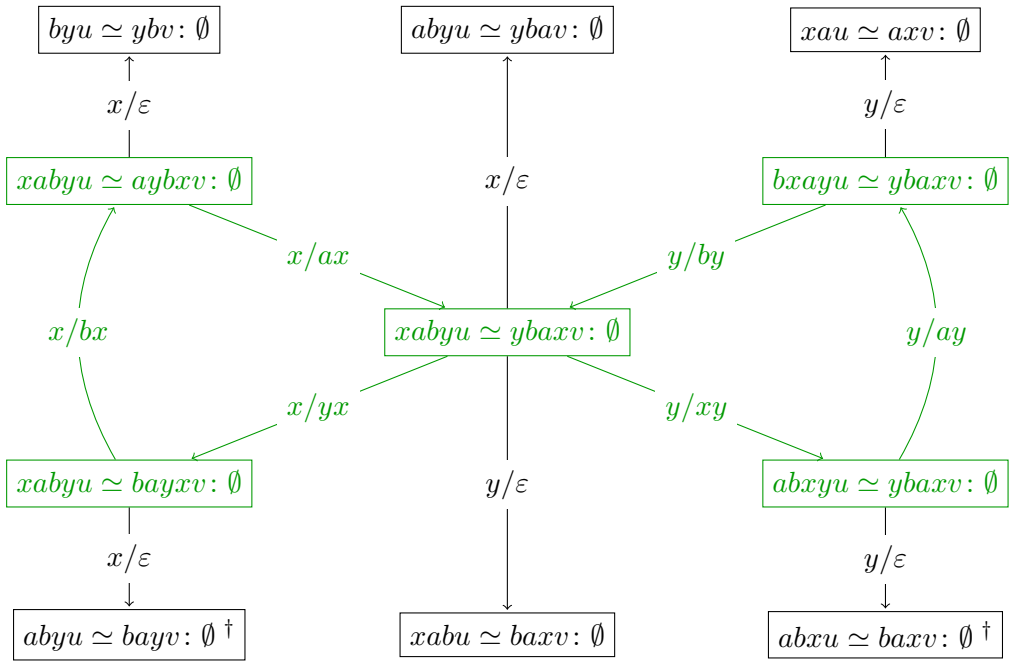


Figure 5.1: Example substitution graph with a bow tie.

The subgraph marked in green is a strongly connected component consisting of two loops. Because of its form, the subgraph is called bow tie. The black nodes are called base cases. A node is called base case if the first edge that points into the node is a substitution of the form x/ε for some $x \in V$. In each base case, one variable is eliminated making the equation easier. This also guarantees that there can be no back-edge from one of the base cases to the bow tie. Similar to section 4.1, there is a strongly connected component and some edges that lead out of it. Like with the loop, it would be beneficial to contract the bow tie into a single node. There would be one edge per base case node with a parameterized substitution σ representing all paths that lead to that node. This would then also work for cases where u or v contain the variables x or y. The only difference would be that in the base case u and v are replaced by $\sigma(u)$ and $\sigma(v)$ respectively.

To see why this is not possible we track the substitution for some possible paths in our example. Starting from $xabyu \simeq ybaxv$ the path takes k_1 iterations on the loop $y/xy, y/ay, y/by$ and the resulting substitution is $\sigma_1(x) = x, \sigma_1(y) = (xab)^{k_1}y$. After k_2 iterations in the loop $x/yx, x/bx, x/ax$ the substitution is $\sigma_2(x) = (yba)^{k_2}x, \sigma_2(y) = ((yba)^{k_2}xaby)^{k_1}$. The path may loop m times alternatingly on the two loops and the resulting substitutions will be increasingly complex nested power expressions. The set of possible substitutions that loop back to the root node is $\bigcup_{n\geq 0}\sigma_m$. It is not possible to represent this set by a single parameterized substitution since the number of parameters depends on m and m can arbitrary high.

5.2 Using Equation Signatures to Identify Patterns

Note how in the example from the previous section the sub-terms u and v did not affect which rules were applied within the bow tie. This holds even if u and v contain the variables x and y.

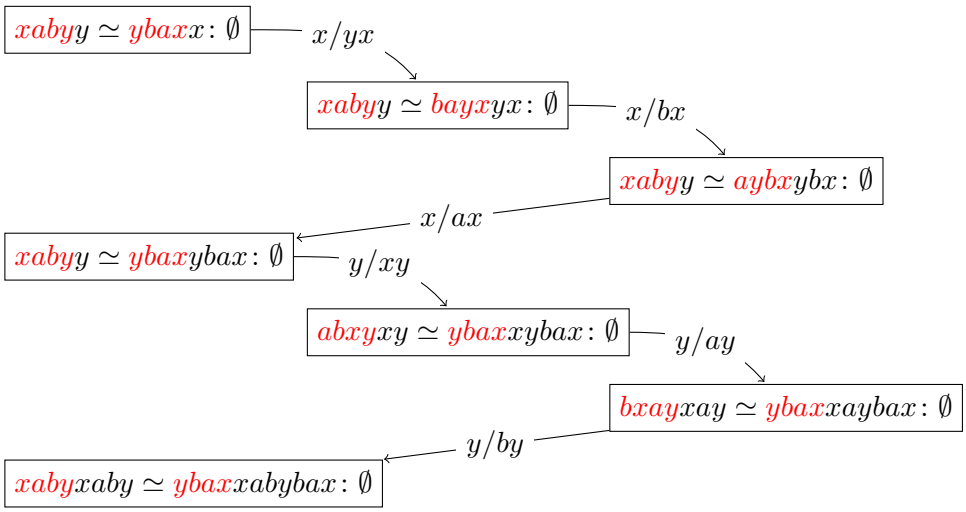


Figure 5.2: Substitution graph with reoccurring prefixes.

In the example of Figure 5.2, u is replaced by y and v by x. In this example, we could infinitely cycle with the substitutions $x/yx, x/bx, x/ax$ as well as $y/xy, y/ax, y/bx$. The first four tokens (in red) will always repeat themselves and are the same as in Figure 5.1. These tokens are called *signature* and determine which rules will be applied. The remaining part will get increasingly longer and prevents that there are loops in the substitution graph. In this example, it would be beneficial to combine the nodes where the signature equal. This way, the graph would have a bow tie, and we would avoid adding infinitely many new nodes.

For any equation $u \simeq v$, the signature is defined as $sig(u \simeq v) = u_1 \simeq v_1$ if and only if there are u_1, u_2, v_1 and v_2 such that u_1 and v_1 are minimal w.r.t. length and satisfy the following conditions:

- $u = u_1u_2$ and $v = v_1v_2$
- u_1 and v_2 are non-empty
- Any variable in u_1 also occurs in v_1 at least once or $v_1 = v$
- Any variable in v_1 also occurs in u_1 at least once or $u_1 = u$

If there are no such u_1, u_2, v_1 and v_2, the signature is defined as $sig(u \simeq v) = u \simeq v$.

Often the signature will be the whole equation. In these cases the algorithm defaults back to the originally described steps from subsection 3.1.3.

The construction of the substitution graph is divided into two steps. In the first step base case nodes are left unprocessed. At the edges that point to a base case all variables except the one that is replaced by ε are replaced by a fresh variable. Thus a substitution x/ε is replaced by σ_i where $\sigma_i(x) = \varepsilon, \sigma_i(y) = y'$ for all $y \in V, y \neq x$.

In this step substitutions, including the modified substitutions that lead to a base case, are only applied to the signature. This makes the remaining parts of both sides irrelevant for the equality check between nodes.

After all other nodes are processed the base case nodes are handled the second step. The sub-terms that were not part of the signature can contain variables that should have been changed by substitutions but where the substitutions were not applied. These variables can not occur in the signature, since all variables in the signature are replaced immediately before the base case node. The remaining variables are now restricted to a language that is defined by the substitutions at any of the paths from the root node to the base case. For every variable $x \in V$ it must hold that $x \in Lang(\{\sigma(x) \mid \sigma = \sigma_1 \circ \cdots \circ \sigma_m\})$ for any path $\sigma_1 \cdots \sigma_m$ from the root node to the base case. This must be one distinct path that is the same for all variables.

The base case node is now solved separately. It is the root node of a different substitution graph. If a solution is found it will be intersected with the restriction on the variables from before to get a solution to the original equation.

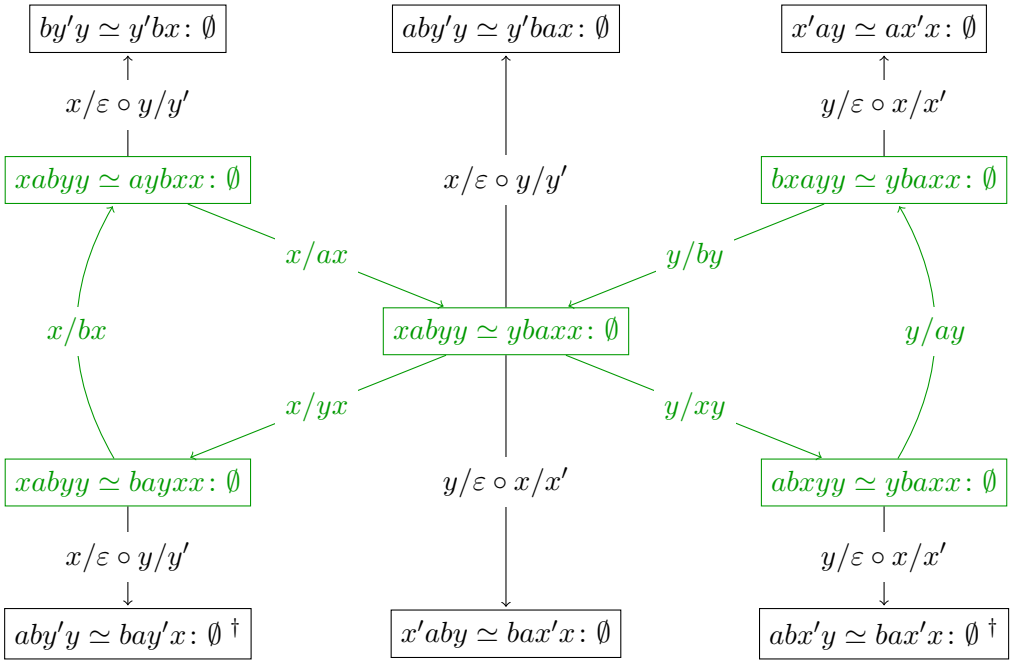


Figure 5.3: Substitution graph using the signature

If no solution is found or the intersection is empty the base case node is a conflict node in the original substitution graph.

5.3 EDT0L Grammars for Capturing Variable Languages

To formalize the variable restrictions a language $\mathcal{L}$ is defined as set of tuples where each element $(x_1, ..., x_k) \in ((V \cup \Sigma)^*) \times \cdots \times ((V \cup \Sigma)^*)$ corresponds to one solution.

To describe this language we use EDT0L grammars [DJK16, SY02, Asv76]. Let $G = (\mathcal{V}_G, \Sigma_G, S, H)$ be a EDT0L grammar where $\mathcal{V}_G$ is a finite set of non-terminals, Σ_G is a finite alphabet of terminals, $S \in V^n$ is a n-tuple of starting non-terminals and H is a set of tables. A table $h \in H$ is a set of morphisms $x \to w$ where $x \in V$ and $w \in (V \cup \Sigma)^*$. Starting from S a table from H will be applied since no non-terminal occurs in the resulting tuple. A table is applied by applying all morphisms to all occurrences of non-terminals of all elements in a tuple.

Let (N, E) be a substitution graph where all occurring variables are given by $V = V_{eq} \cup V'$ where V_{eq} is the set of variables that occur in the input equation and V' the set of fresh variables. Then let $n \in N$ be a base case node. We define a grammar $G_n = (\mathcal{V}_G, \Sigma_G, S, H_n)$. There is a non-terminal for every variable in V_{eq} and every non base case node $N_T \subseteq N$ in the graph $\mathcal{V}_G = \{X_i \mid (x, p_i) \in V_{eq} \times N_T\}$. The terminals $\Sigma_G = \Sigma \cup V'$ are the characters from the alphabet and the fresh variables. The starting

non-terminals are $S = (X_0, Y_0, Z_0, ...)$ where $\{x, y, z, ...\} = V_{eq}$ and $p_0 \in N$ is the root node.

Only edges within the connected component and the edge leading into the base case are considered. For every such edge (n_i, σ, n_j) where $n_i \in N_T$ and $n_j \in N_T \cup \{n\}$ there is a table $h \in H_n$. The table is constructed as $h = \{X_i \to node_j(\sigma(x)) \mid x \in V_{eq}\}$ where $node_j$ replaces every variable $y \in V_{eq}$ with the non-terminal Y_j.

For any base case n of a substitution graph (N, E) all variables must be in the language defined by the corresponding grammar $(x_1, ..., x_k) \in \mathcal{L}(G_n)$.

The grammar that corresponds to Figure 5.3 for the base case $n = x'ay \simeq ax'x$ is

$$G_n = (\{X_0, X_1, X_2, X_3, X_4, Y_0, Y_1, Y_2, Y_3, Y_4\}, \{a, b, x', y'\}, (X_0, Y_0), H_n\}$$

$$\begin{aligned} H_n = \{\{&X_0 \to Y_1X_1, && Y_0 \to Y_1\},\\ \{&X_1 \to bX_2, && Y_1 \to Y_2\},\\ \{&X_2 \to aX_0, && Y_2 \to Y_0\},\\ \{&X_0 \to X_3, && Y_0 \to X_3Y_3\},\\ \{&X_3 \to X_4, && Y_3 \to aY_4\},\\ \{&X_4 \to X_0, && Y_4 \to bY_0\},\\ \{&X_4 \to x', && Y_4 \to \varepsilon\}\} \end{aligned}$$

In the tables in H_n we can reproduce the loops from the graph. We start with the tuple (X_0, Y_0). While in theory every table can be applied only the tables $\{X_0 \to Y_1X_1, Y_0 \to Y_1\}$ and $\{X_0 \to X_3, Y_0 \to X_3Y_3\}$ will change the tuple. We choose the first one and get the new tuple (Y_1X_1, Y_1). Now only table $\{X_1 \to bX_2, Y_1 \to Y_2\}$ is sensible and it will give the tuple (Y_2bX_2, Y_2). Again only one table, namely $\{X_2 \to aX_0, Y_2 \to Y_0\}$ is sensible, it gives (Y_0baX_0, Y_0). Since these three tables can only be applied in this exact sequence they can be replaced by one table that combines them: $X_0 \to Y_0baX_0, Y_0 \to Y_0$. This corresponds to one loop in the substitution graph.

Similarly a table $\{X_0 \to X_0Y_0 \to X_0abY_0\}$ can be introduced for the second loop combining $\{X_0 \to X_3, Y_0 \to X_3Y_3\ \}$, $\{X_3 \to X_4, Y_3 \to aY_4\}$ and $\{X_4 \to X_0, Y_4 \to bY_0\}$.

The sequence $\{X_0 \to X_3, Y_0 \to X_3Y_3\ \}$, $\{X_3 \to X_4, Y_3 \to aY_4\}$ and $\{X_4 \to x', Y_4 \to \varepsilon\}$ is the only sequence of tables that will not lead to a loop but results in a state where all symbols are terminals. This can also be combined to one single table $\{X_0 \to x'Y_0 \to x'a\}$.

With these three newly introduced combined tables all other tables as well as the intermediate non-terminals X_1 to X_4 and Y_1 to Y_4 were rendered needless. We can define a new grammar G'_n that expresses the same language as G_n.

$$G'_n = (\{X_0, Y_0\}, \{a, b, x', y'\}, (X_0, Y_0), H'_n\}$$

$$H'_n = \{\{X_0 \to Y_0baX_0, \quad Y_0 \to Y_0\},$$
$$\{X_0 \to X_0, \quad Y_0 \to X_0abY_0\},$$
$$\{X_0 \to x', \quad Y_0 \to x'a\}\}$$

5.4 Towards a Final Solution to the Equation

Continuing on the example from Figure 5.3 we try to find a solution for the base case $x'ay \simeq ax'x$. After applying rule (4.1) the equation simplifies to $x \simeq y$ giving us the two equivalent solutions $\sigma_1(x) = y, \sigma_1(y) = y$ and $\sigma_1(y) = y, \sigma_1(x) = y$. This means an interpretation satisfies the base case if and only if $\sigma(x) = \sigma(y)$. For the original equation $xabyy \simeq ybaxx$ there is the additional condition that $(\sigma(x), \sigma(y)) \in \mathcal{L}(G_n)$. For any pair $(\sigma(x), \sigma(y)) \in \mathcal{L}(G_n)$ the derivation starts by applying some table $h_0 \in H_n$. A case distinction is made on which table in H_n is h_0.

If $h_0 = \{X_0 \to Y_0baX_0, Y_0 \to Y_0\}$ it follows that $\sigma(x) = ubav$ and $\sigma(y) = u$ where u and v are some string terms that are derivable from Y_0 and X_0 respectively. No matter what u and v are $ubav \neq u$. For the case $h_0 = \{X_0 \to X_0, Y_0 \to X_0abY_0\}$ a symmetric argument can be made. The last case $h_0 = \{X_0 \to x', Y_0 \to x'a\}\}$ gives $\sigma(x) = x'a$ and $\sigma(y) = x'$ where again $\sigma(x) \neq \sigma(y)$.

For all three cases there is no solution σ such that $(\sigma(x), \sigma(y)) \in \mathcal{L}$ and $\sigma(x) = \sigma(y)$. Thus the base case $x'ay \simeq ax'x$ is a conflict node. Similar arguments can be made to show that the remaining base cases are also conflict nodes and consequently show that the equation $xabyy \simeq ybaxx$ is unsatisfiable.

In general the solution for a base case can be a regular language or another EDT0L language. To get a solution for the original equation the intersection of two EDT0L languages or the intersection between an EDT0L language and a regular language is needed. To our knowledge there is no algorithm to compute these intersections without restoring it back to a string equations.

CHAPTER 6

Conclusion

In this thesis we presented the basic idea of solving string equations as well as three prominent approaches. We saw Nielsen Transformation with a set of rules that we extended with our own rules. Using these set of rules we demonstrated how a substitution graph can be used to systematically find models for string equations or prove them unsatisfiable.

We also looked at recompression that simplifies equations systematically by compressing pairs and blocks of characters until the equation is trivially solvable. Many solvers use regular automata to represent and solve string equations and other string constraints. We reviewed one approach using automata called stabilization.

The biggest drawback of Nielsen Transformation is that it might not terminate on unsatisfiable instances outside the quadratic fragment. We successfully used power introduction to overcome this drawback at least in some instances. With these power terms we could represent all solutions, a cycle in the graph would generate, in the form of one parameterized term.

In an attempt to extend our improvements in handling loops to other strongly connected components we introduced signatures. Using these signatures, the problem was split into solving the part of the equation in the signature and, depending on the base case, solving the rest of the equation. With EDT0L grammars we found a fitting representation for these partial solutions.

We think the concept of signatures and the transformation from a substitution graph to an EDT0L grammar are useful tools for devising new approaches to string solving.

This collection of techniques provides a broad insight into the fundamentals of string solving and offers value for solver developers seeking to implement or extend string solvers.

Übersicht verwendeter Hilfsmittel

Für diese Arbeit habe ich keine Textpassagen von KI-Tools übernommen. Die KI-Text-Korrektur von Scribbr in der Version vom 15.08.2025 habe ich verwendet um Grammatik, Rechtschreibung und andere sprachliche Fehler zu finden. ChatGPT mit dem Sprachmodell GPT-4 wurde verwendet, um sprachliche Fehler und unklare Formulierungen aufzuzeigen. Die entsprechenden Stellen habe ich dann ohne KI Hilfsmittel korrigiert.

List of Figures

Bibliography

[AAC+14] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. String constraints for verification. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*, pages 150–166. Springer, 2014.

[Ama23] Roberto Amadini. A survey on string constraint solving. *ACM Comput. Surv.*, 55(2):16:1–16:38, 2023.

[Asv76] {Peter R.J.} Asveld. *A Characterization of ET0L and EDT0L Languages*. Number 129 in Memorandum / Department of Applied Mathematics. University of Twente, Netherlands, 1976.

[BBB+22] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. cvc5: A versatile and industrial-strength smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442. Springer, 2022.

[BCC+23] František Blahoudek, Yu-Fang Chen, David Chocholatỳ, Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Juraj Síč. Word equations in synergy with regular constraints. In *International Symposium on Formal Methods*, pages 403–423. Springer, 2023.

[CCH+24] Yu-Fang Chen, David Chocholatý, Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Juraj Síč. Z3-noodler: An automata-based string solver. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 24–33, Cham, 2024. Springer Nature Switzerland.

[CHL+19] Taolue Chen, Matthew Hague, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.

[Day22] Joel D. Day. Word equations in the context of string solving. In Volker Diekert and Mikhail Volkov, editors, *Developments in Language Theory*, pages 13–32, Cham, 2022. Springer International Publishing.

[DJK16] V Diekert, A Jez, and M Kufleitner. Solutions of word equations over partially commutative structures. 43rd int. In *Colloq. on Automata, Languages, and Programming (ICALP 2016)*, volume 55, page 127, 2016.

[DKM+20] Joel D Day, Mitja Kulczynski, Florin Manea, Dirk Nowotka, and Danny Bøgsted Poulsen. Rule-based word equation solving. In *Proceedings of the 8th International Conference on Formal Methods in Software Engineering*, pages 87–97, 2020.

[dMB08a] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[dMB08b] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[EMS07] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *ISSTA*, pages 151–162. ACM, 2007.

[ESBK25] Clemens Eisenhofer, Theodor Seiser, Nikolaj S. Bjørner, and Laura Kovács. On solving string equations via powers and Parikh images. In *TABLEAUX*, 2025.

[ESM+23] Benjamin Eriksson, Amanda Stjerna, Riccardo De Masellis, Philipp Rümmer, and Andrei Sabelfeld. Black Ostrich: Web application scanning with string solvers. In *SIGSAC*, pages 549–563. ACM, 2023.

[Gut98] Claudio Gutiérrez. Solving equations in strings: On makanin's algorithm. In *Latin American Symposium on Theoretical Informatics*, pages 358–373. Springer, 1998.

[Hag19] Matthew Hague. Strings at MOSCA. *ACM SIGLOG News*, 6(4):4–22, 2019.

[HRS19] Hossein Hojjat, Philipp Rümmer, and Ali Shamakhi. On strings in software model checking. In *Asian Symposium on Programming Languages and Systems*, pages 19–30. Springer, 2019.

[Jeż16] Artur Jeż. Recompression: a simple and powerful technique for word equations. *Journal of the ACM (JACM)*, 63(1):1–51, 2016.

[LM21] Anthony W Lin and Rupak Majumdar. Quadratic word equations with length constraints, counter systems, and presburger arithmetic with divisibility. *Logical Methods in Computer Science*, 17, 2021.

[LRT+14] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. A dpll (t) theory solver for a theory of strings and regular expressions. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*, pages 646–662. Springer, 2014.

[Mak77] Gennadiy Semenovich Makanin. The problem of solvability of equations in a free semigroup. *Matematicheskii Sbornik*, 145(2):147–236, 1977.

[MBK+21] Federico Mora, Murphy Berzish, Mitja Kulczynski, Dirk Nowotka, and Vijay Ganesh. Z3str4: A multi-armed string solver. In *FM*, volume 13047 of *LNCS*, pages 389–406. Springer, 2021.

[Nie17] Jakob Nielsen. Die isomorphismen der allgemeinen, unendlichen gruppe mit zwei erzeugenden. *Mathematische Annalen*, 78(1):385–397, 1917.

[Pla99] Wojciech Plandowski. Satisfiability of word equations with constants is in pspace. In *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*, pages 495–500. IEEE, 1999.

[Rüm08] Philipp Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In *LPAR*, volume 5330 of *LNCS*, pages 274–289. Springer, 2008.

[SY02] Kai Salomaa and Sheng Yu. Decidability of edt0l structural equivalence. *Theoretical computer science*, 276(1-2):245–259, 2002.

[TCJ14] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *SIGSAC*, pages 1232–1243. ACM, 2014.

[WS07] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *SIGPLAN*, pages 32–41. ACM, 2007.