Inter-theory Dependency Analysis for SMT String Solvers

MINH-THAI TRINH, Advanced Digital Sciences Center, Illinois at Singapore, Singapore
DUC-HIEP CHU*, National University of Singapore, Singapore
JOXAN JAFFAR, National University of Singapore, Singapore

Solvers in the framework of Satisfiability Modulo Theories (SMT) have been widely successful in practice. Recently there has been an increasing interest in solvers for string constraints to address security issues in web programming, for example. To be practically useful, the solvers need to support an expressive constraint language over *unbounded* strings, and in particular, over string lengths. Satisfiability checking for these formulas, especially in the SMT context, is very hard; it is generally undecidable for a rich fragment. In this paper, we propose a form of dependency analysis for a rich fragment of string constraints including high-level operations such as `length`, `contains` to deal with their inter-theory interaction so as to solve them more efficiently. We implement our dependency analysis in the string theory of the Z3 solver to obtain a new one, called `S3N`. Finally, we demonstrate the superior performance of `S3N` over state-of-the-art string solvers such as Z3str3, CVC4, S3P, and Z3 on several large industrial-strength benchmarks.

CCS Concepts: • **Theory of computation** → **Automated reasoning**; • **Security and privacy** → *Web application security*.

Additional Key Words and Phrases: Automated Reasoning, String Constraints, SMT, Inter-theory, Dependency Analysis, Web Security

ACM Reference Format:
Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2020. Inter-theory Dependency Analysis for SMT String Solvers. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 192 (November 2020), 27 pages. https://doi.org/10.1145/3428260

1 INTRODUCTION

Automated reasoning of string constraints has recently drawn a lot of attention. Their efficient reasoning is essential to the growing number of security and verification problems. For example, security analysis of web applications [Saxena et al. 2010; Trinh et al. 2014] usually requires reasoning over them. This is because of the fact that strings are ubiquitous in web applications—a web application usually takes string values as input, manipulates them, and then uses them to construct database queries. More importantly, the improper use of strings usually affects the software security such as the cases of SQL injection and Cross Site Scripting (XSS) flaws (see [OWASP 2013]).

To be practically useful for these analyses, solvers need to support at least the core constraint fragment over *unbounded* strings that includes string equations and length constraints [Reynolds et al. 2017; Saxena et al. 2010; Trinh et al. 2014]. In practice, existing solvers also support extended string constraints, allowing a richer language of string terms over operators such as `contains`,

*Now at Google.

Authors' addresses: Minh-Thai Trinh, Advanced Digital Sciences Center, Illinois at Singapore, 1 Create Way, Create Tower, Singapore, 138602, Singapore, minhthai.t@adsc-create.edu.sg, trinhmt@illinois.edu; Duc-Hiep Chu, National University of Singapore, Singapore; Joxan Jaffar, National University of Singapore, Singapore.

2475-1421/2020/11-ART192
https://doi.org/10.1145/3428260

`index_of` and `replace`, by relying on (eager or lazy) reductions to the constraints of the core fragment.

For a system of only string equations, state-of-the-art SMT string solvers [Berzish et al. 2017; Liang et al. 2014; Reynolds et al. 2017; Trinh et al. 2014, 2016; Zheng et al. 2015] follow an approach rooted in the very first *decision procedure* by Makanin [Makanin 1977]. These deal with string equations by applying reduction rules in order to reach the state where equations are in the SSA form which means one side is an eliminable variable. Therefore, we can derive satisfying assignments for the eliminable variables (based on the other side of equations) without being inconsistent with other constraints. Specifically, solvers can repeatedly apply the so-called *splitting rule*: choosing a string equation in the formula to reduce to simpler equations based on possible *alignments* among the relevant variables. This step often introduces branches and new string variables, but with the hope that eventually either the input formula will be reduced into SSA-form formulas; thus satisfying assignments can be generated, or an inconsistency among string equations is detected.

Unfortunately, this cannot be easily achieved when the system *also* contains length and extended string constraints, which are of *different* sorts. Although the satisfiability problem for string equations is decidable [Makanin 1977], the decidability for the core fragment (i.e. string equations with length constraints) is unknown [Ganesh et al. 2013] and the satisfiability problem for the richer fragment (e.g. with `replace` operation) is even undecidable [Bjørner et al. 2009; Büchi and Senger 1990]. This means that checking the satisfiability of string equations with length (and extended string) constraints is *very hard*. For example, one may continue the search without realizing that the string equations in the current context are already inconsistent with length constraints.

It is well-known that the difficulty comes from the interaction between constraints. In essence, although we can find a satisfying assignment for a variable in one constraint, it might be possible that this assignment does not satisfy other constraints sharing that variable. Therefore, the search algorithm (to find a solution) becomes less efficient because it needs to backtrack in such case. For a system of only string equations, the dependency will be resolved in the solving process because the splitting rule will help to break the formulas into SSA-form formulas, where one side is some eliminable variable. For the core fragment (or richer ones), the interaction is not only among string equations. It is undoubtedly more intricate because the interaction is also among constraints of *different sorts* (e.g. string, integer). We call it the *inter-theory* interaction/dependency. This kind of dependency unfortunately cannot be effectively handled by the above solving process.

In fact, solving string equations along with length constraints and extended string constraints becomes even more challenging in the setting of SMT solvers. This is because, in SMT solvers, constraints of different sorts will be handled by different theory solvers [de Moura and Bjørner 2008; Liang et al. 2014; Trinh et al. 2014; Zheng et al. 2013], each of which usually works independently/separately from others. From the point of view of the solver design (and also efficiency/performance), implementing disjoint theory solvers is of course an advantage of SMT solvers. However, with respect to the *multi-sorted* theories (e.g. string, sequence, set, multi-set), this restriction really raises a big challenge for efficient implementation because of the unavoidable interaction between constraints from different theories.

In this paper, we propose a form of dependency analysis to address the fundamental problem of inter-theory dependency among constraints. In particular, we focus on how to efficiently implement the analysis in the SMT context. At the heart is a measure that takes into account both the complexity of the dependency (i.e. how complicated the dependency is) and the complexity of the solving process (i.e. how much effort to resolve such dependency). Specifically, we aim to resolve the most complicated dependency while minimizing the number of solving "steps". This leads to the construction of a *partial order* in which string equations should be solved in order to break their dependency with length constraints and extended string constraints. The core idea is to solve the

equation that can *most easily* be reduced into the so-called solved form—an SSA-like form—in order to resolve *as many as possible* length constraints and extended constraints. In fact, choosing a good order among the equations can also have a significant impact on how *quickly* the interaction can be resolved. According to this order, string equations that interact with length (and extended string) constraints will come first while the remaining are not ordered and at the end (of the queue). The analysis thus prioritizes satisfying as many as possible length (and extended) constraints, which dually enables us to detect conflicts (if any) with these constraints earlier and learn better conflict clauses among constraints from different theories.

Finally, we implement our technique inside the string/sequence solver of Z3 [de Moura and Bjørner 2008] to obtain a new solver, called S3N. To demonstrate the superior performance of S3N, we comprehensively evaluate it against state-of-the-art SMT string solvers Z3str3 [Berzish et al. 2017], CVC4 [Reynolds et al. 2019, 2017], S3P [Trinh et al. 2016], and also Z3 on two large industrial-strength benchmarks, which are generated by symbolic execution of web applications. For the SAT benchmarks (all constraints are satisfiable), S3N is at least 8 times faster than the solvers which have almost similar robustness. For the UNSAT benchmarks, the speed-up of S3N is not as marked (at least 2 times faster) in comparison with the second best solver, when each input benchmark can be completely solved (proved unsatisfiable) by both solvers. However, the improvement that S3N brings is in the *number* of benchmarks that S3N can decide unsatisfiability. In the end, S3N is the best solver over all the benchmarks.

To summarize, our technical contribution is a dependency analysis for string constraints. In the SMT setting, the analysis is implemented via a partial order for solving string equations. This order is decided based on:

- the interaction of string equations with length constraints and extended string constraints
- the novel solved form distance, which informally measures "how far" they are from their corresponding solved form; the shorter the distance is, the faster relevant length and extended string constraints can be discharged.

We organize the rest of the paper as follows. We give different examples to highlight the limitation of state-of-the-art string solvers and to motivate the introduction of our technique in Section 2. We present the core constraint language and show why it is sufficient to handle practical constraints coming from reasoning about web applications in Section 3. We formalize our dependency analysis in Section 4 and present its implementation inside an SMT solver in Section 5. We show the experimental evaluation of our solver in Section 6. We discuss related work in Section 7 and conclude the paper in Section 8.

2 MOTIVATION

In this section, we start with a simple example to illustrate the use of the splitting rule, which is the fundamental rule to deal with string equations in SMT solvers. Then we discuss how SMT solvers deal with length constraints and extended string constraints in order to highlight their limitation and motivate the introduction of our technique.

Example 2.1 (String Equations). $Z_1 \cdot Z_2 \cdots Z_n = T_1 \cdot T_2 \cdots T_m \wedge F$

Consider the above constraints, where $\cdot$ denotes the string concatenation operator and F is a formula that might involve variables Z_i $(1 \leq i \leq n)$, T_j $(1 \leq j \leq m)$ and other variables. One typical way to proceed is to perform case splitting on the equation $Z_1 \cdot Z_2 \cdots Z_n = T_1 \cdot T_2 \cdots T_m$. There are different ways to split the string variables. For example, w.r.t. the alignment between Z_1 and T_1[1] (i.e. based on the relationship between the length of Z_1 and the length of T_1), there are 3 cases :

[1]We can also split based on the alignment between Z_n and T_m.

- $\texttt{len}(Z_1) = \texttt{len}(T_1)$,
- $\texttt{len}(Z_1) > \texttt{len}(T_1)$,
- $\texttt{len}(Z_1) < \texttt{len}(T_1)$.

Correspondingly, there are 3 disjuncts:

- $Z_1 = T_1 \wedge Z_2 \cdots Z_n = T_2 \cdots T_m \wedge F$
- $Z_1 = T_1 \cdot Z_0 \wedge Z_0 \cdot Z_2 \cdots Z_n = T_2 \cdots T_m \wedge F$
- $T_1 = Z_1 \cdot T_0 \wedge Z_2 \cdots Z_n = T_0 \cdot T_2 \cdots T_m \wedge F$

where Z_0 and T_0 are fresh string variables.

Next, we may choose *one* case to proceed. We note that by applying the splitting rule we might introduce new string variables (e.g., Z_0 and T_0), yet the formulas are simpler because we reduce the number of variables involved in each equation. More importantly, the application of the splitting rule creates a branching point, which enables us to backtrack the search when needed. If we repetitively apply the splitting rule to the equations in one of the disjuncts above, we will finally obtain the formulas in the SSA form which is similar to the form of the first equations in the 3 disjuncts. □

Now let us continue with examples of solving string equations with length constraints and extended string constraints. The next example shows how state-of-the-art SMT solvers solve string equations and length constraints together.

Example 2.2 (String Equations and Length Constraints).

$$X \cdot \text{“}a\text{”} \cdot Y = Z \cdot T \wedge X \cdot \text{“}a\text{”} \cdot Y = X_1 \cdot \text{“}b\text{”} \cdot Y_1 \wedge \texttt{len}(X) = \texttt{len}(X_1)$$

In addition to two string equations, we also have a length constraint $\texttt{len}(X) = \texttt{len}(X_1)$. Starting with the given formula, SMT solvers can proceed by applying the splitting rule either to the first equation $X \cdot \text{“}a\text{”} \cdot Y = Z \cdot T$ or to the second equation $X \cdot \text{“}a\text{”} \cdot Y = X_1 \cdot \text{“}b\text{”} \cdot Y_1$. However, knowing that $\texttt{len}(X) = \texttt{len}(X_1)$, SMT solvers will apply the splitting rule to the second equation, reducing the formula to only *one* possible subcase

$$X \cdot \text{“}a\text{”} \cdot Y = Z \cdot T \wedge X = X_1 \wedge \text{“}a\text{”} \cdot Y = \text{“}b\text{”} \cdot Y_1$$

The solvers then can immediately declare unsatisfiability due to the conflict between “a” and “b” in the last equation. □

Although it looks very easy for solvers to deal with length constraints in the above example, solving string equations with length constraints is in general a very hard problem. State-of-the-art string solvers (*including* SMT-based solvers) can only handle either a class of length constraints [Ganesh et al. 2013] or a class of string equations [Lin and Majumdar 2018], but *not* the whole together. Basically, at first glance, one can think of a solution by implementing inference rules to solve string equations and length constraints simultaneously. However, these rules usually have to be based on the syntax of length constraints and thus very *ad-hoc*. To the best of our knowledge, there is no systematic solution to this problem yet.

As stated before, this problem becomes even harder in the SMT context, where there is a separation between two theories: string and integer. Specifically, we cannot directly implement inference rules to solve string equations and length constraints simultaneously. Instead, in the above example, SMT solvers have to query the length information from the integer solver so that the string solver can use it; see Section 5 for further details. In general, making use of length constraints when solving string equations is still a big challenge for SMT solvers.

In short, what we need are i) a systematic method to handle string equations with length constraints and ii) an efficient implementation of this method in the setting of SMT solvers. Such need is also applicable to extended string constraints, which is illustrated via the following example.

Example 2.3 (String Equations and Extended String Constraints).

$$X_1 \cdot Y_1 \cdot Z_1 = X_2 \cdot \text{“}a\text{”} \cdot Z_2 \wedge Y_1 \cdot Z_2 = \text{“}a\text{”} \cdot Y_2 \wedge \neg\texttt{contains}(Z_2, \text{“}a\text{”})$$

In this example, there is also an extended constraint $\neg\texttt{contains}(Z_2, \text{“}a\text{”})$, meaning Z_2 does not contain “a”. Existing SMT solvers reason about it by working on the two string equations, finding a more concrete form for Z_2 before checking against the extended constraint. This is because if we reduce the constraint $\neg\texttt{contains}()$, we usually need to introduce universal quantifiers, which lead to a more difficult problem.

In Example 2.3, we can see that both the two string equations share the variable Z_2 with the constraint $\neg\texttt{contains}(Z_2, \text{“}a\text{”})$. So both the two equations depend on the extended string constraints. Choosing which equation to solve first in order to break the dependency is very important. If a solver applies the splitting rule to the first equation, it can obtain a disjunctive formula based on the alignment between X_1 and X_2 as follows.

$$(\texttt{len}(X_1)<\texttt{len}(X_2) \wedge X_2{=}X_1 \cdot T \wedge Y_1 \cdot Z_1{=}T \cdot \text{“}a\text{”} \cdot Z_2 \wedge Y_1 \cdot Z_2{=}\text{“}a\text{”} \cdot Y_2 \wedge \neg\texttt{contains}(Z_2, \text{“}a\text{”}))$$

$$\vee \quad (\texttt{len}(X_1)\geq\texttt{len}(X_2) \wedge X_1{=}X_2 \cdot X_3 \wedge X_3 \cdot Y_1 \cdot Z_1{=}\text{“}a\text{”} \cdot Z_2 \wedge Y_1 \cdot Z_2{=}\text{“}a\text{”} \cdot Y_2 \wedge \neg\texttt{contains}(Z_2, \text{“}a\text{”}))$$

Suppose it proceeds with the second disjunct. Next, it can perform splitting on the second equation $X_3 \cdot Y_1 \cdot Z_1{=}\text{“}a\text{”} \cdot Z_2$ to have a new disjunctive formula:

$$(X_1{=}X_2 \cdot X_3 \wedge X_3{=}\text{“”} \wedge Y_1 \cdot Z_1{=}\text{“}a\text{”} \cdot Z_2 \wedge Y_1 \cdot Z_2{=}\text{“}a\text{”} \cdot Y_2 \wedge \neg\texttt{contains}(Z_2, \text{“}a\text{”}))$$

$$\vee \quad (X_1{=}X_2 \cdot X_3 \wedge X_3{=}\text{“}a\text{”} \cdot X_4 \wedge X_4 \cdot Y_1 \cdot Z_1{=}Z_2 \wedge Y_1 \cdot Z_2{=}\text{“}a\text{”} \cdot Y_2 \wedge \neg\texttt{contains}(Z_2, \text{“}a\text{”}))$$

In the second disjunct, it can apply the splitting rule to the last equation to obtain 2 subcases:

- when Y_1 is an empty string and
- when Y_1 is “a” $\cdot\, Y_3$ with $\texttt{len}(Y_3)\geq 0$.

It now finds out that there is a conflict in both cases.

- If $Y_1 =$ “” then there is a conflict between $Z_2 =$ “a” $\cdot\, Y_2$ and $\neg\texttt{contains}(Z_2, \text{“}a\text{”})$.
- If $Y_1 =$ “a” $\cdot\, Y_3$ then there is a conflict between $Z_2 = X_4 \cdot Y_1 \cdot Z_1$ and $\neg\texttt{contains}(Z_2, \text{“}a\text{”})$.

In both cases, it has to backtrack and search for a solution in other branches.

In contrast, given the input formula, our algorithm applies the splitting rule to the second equation first. Specifically, we will have 2 cases:

- when Y_1 is an empty string and
- when Y_1 is “a” $\cdot\, Y_3$, with $\texttt{len}(Y_3)\geq 0$

where the second case will lead to a satisfiable solution. Even if we choose to proceed with the first case, the cost of backtracking is still much less expensive since we can quickly detect a conflict between $Z_2 =$ “a” $\cdot\, Y_2$ and $\neg\texttt{contains}(Z_2, \text{“}a\text{”})$. □

A hint is that if we look closer at the original formula in Example 2.3, we can see that if we apply the splitting rule to the second equation $Y_1 \cdot Z_2 =$ “a” $\cdot\, Y_2$, we can more quickly reach the state where the value of Z_2 can be easily derived. This helps us break the inter-theory dependency by discharging the constraint $\neg\texttt{contains}(Z_2, \text{“}a\text{”})$ quicker. We will formalize our dependency analysis in the following sections.

3 CONSTRAINT LANGUAGE

In Figure 1, we introduce the core constraint language which includes string equations, length constraints, and irreducible extended string constraints. Then we will explain how we can reduce other kinds of string constraints into this core language.

$$
\begin{array}{lcll}
Fml & ::= & Literal & \\
 & | & \neg\, Literal & \\
 & | & Fml \wedge Fml & \\
Literal & ::= & A_{str} & \\
 & | & A_{len} & \\
 & | & A_{ext} & \\
A_{str} & ::= & T_{str} = T_{str} & \\
A_{len} & ::= & T_{len} \leq m & (m \in C_{int}) \\
T_{str} & ::= & a & (a \in C_{str}) \\
 & | & X & (X \in V_{str}) \\
 & | & T_{str} \cdot T_{str} & \\
 & | & T_{ext} & \\
T_{ext} & ::= & \mathsf{Func}(\tilde{T}_{str}, \tilde{T}_{reg}, \tilde{T}_{len}) & \\
T_{reg} & ::= & a & (a \in C_{str}) \\
 & | & (T_{reg})^{\star} & \\
 & | & T_{reg} \cdot T_{reg} & \\
 & | & T_{reg} + T_{reg} & \\
T_{len} & ::= & m & (m \in C_{int}) \\
 & | & M & (M \in V_{int}) \\
 & | & \mathtt{len}(T_{str}) & \\
 & | & \Sigma_{i=1}^{n}(m_i * T_{len}) & (m_i \in C_{int})
\end{array}
$$

Fig. 1. The Syntax of Our Core Constraint Language

3.1 Core Language

Variables: There are two types of variables: V_{str} consists of string variables; and V_{int} consists of integer variables. We use capital letters to denote variables.

Constants: Correspondingly, there are two types of constants: string and integer constants. Let C_{str} be a subset of $\Sigma^{\star}$ for some finite alphabet Σ. Elements of C_{str} are referred to as constant strings. Elements of C_{int} are integers. We use small letters to denote constants when not referring to their concrete values.

Terms: A term (or expression) (denoted D, E, and possibly with subscripts) is either a length term, a string term or a regular expression term.

- A length term T_{len} is an element of V_{int}, an element of C_{int}, `len` function applied to a string term, a constant integer multiple of a length term, or their sum. We note that although in Figure 1, we only mention linear arithmetic constraints, our language also supports other kinds of integer constraints. Specifically, whatever a built-in integer solver can support, we can also support since we implement our technique inside an SMT solver.
- A string T_{str} term is either an element of C_{str}, an element of V_{str}, a concatenation of terms (using $\cdot$ operator), or a function on terms, which is abstracted as an "extended" string term T_{ext}. We will explain this extended term in detail later.
- A regular expression term T_{reg} is constructed from string constants by using operators such as concatenation ($\cdot$), union ($+$), and Kleene star ($\star$).

Literals: A literal is either a string equation (A_{str}), a length constraint (A_{len}) or an irreducible extended string constraint (A_{ext}). An extended string constraint A_{ext} is just a function like T_{ext}, except that it returns a boolean value (instead of a string value).

Formulas: Formulas (denoted F, G, H, I, and possibly with subscripts) are defined inductively over literals by using operators such as conjunction ($\wedge$) and negation ($\neg$). Note that, each theory solver of SMT solvers considers only a conjunction of literals at a time. The disjunction will be handled by the core SAT engine. We use $\texttt{Var}(F)$ to denote the set of all variables of F, including bound variables.

Language: Finally, we define L to be the quantifier-free first-order two-sorted language over which the formulas described above are constructed. In essence, the language contains string equations, length constraints, and irreducible extended string constraints.

3.2 Other String Constraints

To sufficiently reason about web applications, modern string solvers need to support formulas of quantifier-free first-order logic over string equations, length constraints, membership predicates, and also high-level string operations from programming languages such as JavaScript and Python [Reynolds et al. 2017; Saxena et al. 2010; Trinh et al. 2014]. Thus we will describe the reduction from other constraints, i.e. membership predicates and high-level string operations, to the core language.

We follow the same approach of [Trinh et al. 2014, 2016; Zheng et al. 2015], reducing membership predicates into string equations where Kleene star operations are represented as recursive `star` functions. The `star` function takes two parameters as its input. The first parameter is a regular expression term while the second is a non-negative integer variable. For example, $X \in (r)^\star$ is modeled as $X = \texttt{star}(r, N)$, where N is a *fresh* variable denoting the number of times that r is repeated. For other kinds of regular expression terms, we recursively reduce them as follows:

$$\begin{array}{lcl} X \in a & \rightarrow & X = a \\ X \in (r_1 + r_2) & \rightarrow & X \in r_1 \vee X \in r_2 \\ X \in (r_1 \cdot r_2) & \rightarrow & \exists X_1, X_2 :\ X = X_1 \cdot X_2 \wedge X_1 \in r_1 \wedge X_2 \in r_2 \end{array}$$

Note that we do *not* introduce existential variables (e.g. X_1, X_2). Instead, we use Skolem functions to represent them. We use existential variables in this paper for a shorter presentation only. Intuitively, if the RHS is a string constant, we can reduce the membership predicate into a string equation. If the RHS is a union, we can reduce the predicate by splitting the union into two cases. If the RHS is a concatenation, we can reduce the LHS into two parts, each of which satisfies the corresponding membership predicate. If there is no Kleene star operation in a membership predicate $X \in r$, it is clearly that the above reduction rules can completely reduce the membership predicate to string equations. The `star` function will not be completely reduced and be one instance of T_{ext} in Figure 1.

For high-level string operations, most of them can be completely reduced into string equations and length constraints. For example,

$$\begin{array}{lcl} \texttt{contains}(X, a) & \rightarrow & \exists X_1, X_2 : X = X_1 \cdot a \cdot X_2 \\ Y = \texttt{charAt}(X, m) & \rightarrow & \exists X_1, X_2 :\ X = X_1 \cdot Y \cdot X_2 \wedge \texttt{len}(X_1) = m \wedge \texttt{len}(Y) = 1 \end{array}$$

given $0 \leq m \leq \texttt{len}(X)$. However, since $\neg\texttt{contains}$ is not reduced as shown in Section 2, other operations relying on that are not completely reduced into string equations and length constraints.

$$\begin{array}{lcl} M = \texttt{indexOf}(X, a) & \rightarrow & (\neg\texttt{contains}(X, a) \ \wedge \ M = -1) \\ & \vee & \exists X_1, X_2 : (X = X_1 \cdot a \cdot X_2 \wedge \neg\texttt{contains}(X_1, a) \wedge M = \texttt{len}(X_1)) \end{array}$$

In addition, other operations such as `replace`, `match` are also not completely reduced. For example, the `replace` function will be recursively defined as follows.

$$\begin{array}{rcl} Y = \texttt{replace}(X, r, Z) & \rightarrow & (X \notin (.^\star\ r\ .^\star)\ \wedge\ Y = X) \\ & \vee & \exists X_1, X_2, X_3, X_4, Y_1 : (X{=}X_1{\cdot}X_2{\cdot}X_3{\cdot}X_4\ \wedge\ X_2{\cdot}X_3 \in (r)\ \wedge\ \texttt{len}(X_3){=}1 \\ & & \wedge\ X_1 \cdot X_2 \notin (.^\star\ r\ .^\star)\ \wedge\ Y = X_1 \cdot Z \cdot Y_1\ \wedge\ Y_1 = \texttt{replace}(X_4, r, Z)) \end{array}$$

We refer interested readers to [Reynolds et al. 2017; Saxena et al. 2010; Trinh et al. 2014, 2016; Zheng et al. 2015] for the reduction of other string operations.

To summarize, the irreducible parts appear in two places in Figure 1. These are T_{ext} and A_{ext}. The main difference between them is just the sort of the return result; T_{ext} returns a string value while A_{ext} returns a boolean value. An example of T_{ext} is the `replace` function while an example of A_{ext} is `contains` function (under the negation).

4 INTER-THEORY DEPENDENCY ANALYSIS

Before introducing our dependency analysis, let us quickly recall how the string theory solver works as part of a DPLL(T) framework, which contains multiple built-in solvers to deal with constraints from different theories. More details will be presented in Section 5.

As explained in Section 3, the core constraints of interest include string equations, length constraints and extended string constraints. Because the first and last one are string constraints, these are handled by a string solver. Meanwhile, the length constraints are handled by the integer theory solver. Basically, there will be some interaction between the two theory solvers. However, such interaction in general is very limited because of the separation between the two theories (see Section 5).

For string constraints, the search for a solution is basically driven by a set of *derivation rules*.

Definition 4.1 (Derivation Rule). *Each rule is of the general form*

$$\textit{(RULE-NAME)}\quad \frac{F}{\bigvee_{i=1}^{m} G_i}$$

where F, G_i are formulas, $F \equiv \bigvee_{i=1}^{m} G_i$, and $\texttt{Var}(F) \subseteq \texttt{Var}(G_i)$. □

Applying an instance of this (template) rule transforms the formula F at the top into the formula at the bottom, which comprises m *reducts* G_i.

At the high level, we can ignore the classification of constraints into different theories. So we can abstract the search for a solution via the construction of a derivation tree for the original formula by applying a set of derivation rules. The derivation tree for each formula F is formalized as follows. (Illustrative examples will be presented later.)

Definition 4.2 (Derivation Tree). *A derivation tree for a formula F is obtained by applying a derivation rule R to F. Let the reducts of R be G_i, $1 \leq i \leq m$. Then the tree comprises a root labelled with F and m child nodes, labelled with G_i, $1 \leq i \leq m$.* □

A derivation tree rooted at the original formula is constructed using some search strategy. The commonly-used one is a form of Depth First Search. In navigating the construction of the derivation tree, we backtrack when we encounter a `false` formula. If all the leaf nodes of a subtree rooted at F are `false`, we can decide the formula F is unsatisfiable.

4.1 A Quick Overview

Formulas that arise from reasoning of web programs often involve a large number of string equations, which are in the interaction with length constraints and extended string constraints. To

address the fundamental problem of inter-theory dependency among these constraints, we propose a form of dependency analysis, which in particular, is efficiently implemented in SMT solvers as described in Section 5.

We first define a measure that takes into account both the complexity of the dependency (i.e. how complicated the dependency is) and the complexity of the solving process (i.e. how much effort to resolve such dependency). Specifically, we aim to resolve the most complicated dependency while minimizing the number of solving steps. We start by "solving" each string equation, independently of the other string equations, yet taking into account the dependency with length constraints and extended constraints; such dependency is described via the priority value for each string variable. This (local) attempt gives us an estimate of how easily the equation can be reduced to the so-called *solved form* (defined later)—an SSA-like form—and also a *witness* path, which is described via a sequence of *branching decisions* (also defined later), to achieve that. Note that this attempt applies to a single equation, thus its cost is *much cheaper*.

This leads to the construction of a *partial order* in which string equations should be solved in order to break their dependency with length constraints and extended string constraints. The idea is to solve the current formula by applying the splitting rule to the equation that can *most easily* be reduced into solved form in order to resolve *as many as possible* length constraints and extended constraints. The logic behind this will be explained later. In fact, choosing a good order among the equations can also have a significant impact on how *quickly* the interaction can be resolved. There are four important properties that give rise to the efficiency of our method:

- Since we can quickly achieve the state where variables of interest appear in solved form formulas, the relevant length and extended string constraint can easily be dealt with. This will help us break the (inter-theory) dependency between string equations and length (and extended) constraints more easily.
- This procedure is in line with the existing procedures for solving string equations, where the reduction rules will finally lead us to formulas that are in solved form or the alike.
- The local attempt, i.e. solving each equation separately, is very cheap.
- Our recorded branching decisions from solving *locally* an equation can be directly replayed (a form of reuse) in the context of the original (*global*) formula.

4.2 Partial Order for String Equations

The complexity of solving each string equation independently is estimated via a novel *solved-form distance*. When computing a distance, we also collect the corresponding list of branching decisions, which serves as a witness. This does require us construct a derivation tree. However, the construction is controllable by applying only the first applicable rule from our orderly fixed set of rules (at any node). (The selective rules are presented in Section 5.)

Definition 4.3 (Solved Form). *Given a formula F and a string variable X that appears in F, F is said to be in* solved form *wrt. X if there exists exactly one string equation of the form $X = E$ (or $E = X$) in F and X does not appear in E.* □

Since this is a key definition, we would like to clearly explain the corner cases via three simple examples. In the first example, suppose $F \equiv G \wedge X = E$ is in solved form wrt. X. Now we can eliminate X in G by substituting all X in G with E to obtain a new formula G'. Obviously, $X \notin \texttt{Var}(G')$. This helps us achieve a simpler formula G' since we have eliminated its variable X. In addition, for every string variable Y appearing in E, if $Y \notin \texttt{Var}(G')$, we also consider F to be in solved form wrt. Y. This is because if we introduce a fresh variable Z such that $Y = Z$ then Def. 4.3 can be applied to $F \equiv G \wedge X = E \wedge Y = Z$ and the string variable Y. In short, what we have is that (the value of) X depends on (the value of) Y and Y depends on Z.

As another example, the formula $X = Y \wedge Y = X \cdot a$ is in solved form wrt. X but not wrt. Y since we have *two* constraints $Y = X$ and $Y = X \cdot a$. Note that when a formula is in solved form wrt. one variable, it does not guarantee the satisfiability of the formula. The meaning of a solved form wrt. X in this example is that if we can find a satisfying assignment for Y, then we can also find a satisfying assignment for X. Of course, in this case we cannot find any satisfying assignment for Y; thus the formula is unsatisfiable.

The third example is about the so-called *overlapping variable*. This happens when a variable appears in both sides of a string equation, for example, $X \cdot a = b \cdot X$. The important thing here is that such kind of formula cannot be transformed into a solved-form formula. If we continue to apply the splitting rule to this equation, we will go into an infinite loop. In our implementation, since we only explore the derivation tree up to a certain depth as described in Section 5, this is not an issue; we will revisit this point at the end of Section 5. In general, we can make use of previous techniques such as [Trinh et al. 2016; Zheng et al. 2015] to handle the overlapping variables.

The solved form does not only allow us to eliminate variables. It also helps us simplify string equations and length constraints in order to i) make it easier to detect conflicts (if any) between string equations and length constraints and ii) allow the splitting rules with length constraints to be applicable and thus the length constraints can be discharged. For example, suppose we have $X_1 \cdot X_2 \cdot X_3 = Y_1 \cdot Y_2 \cdot Y_3$ and a length constraint $F \equiv 2 * \texttt{len}(X_1) = 2 * \texttt{len}(Y_1) + \texttt{len}(Y_2)$, along with other string equations and length constraints. Currently, we have no rule to handle such complicated constraints like F. If we can reach the solved form for X_1, for example, depending on which branch we follow, we can either find a conflict with F and then learn a conflict clause, that is, $\texttt{len}(X_1) \geq \texttt{len}(Y_1)$, or we can simplify the current formula into

$$Z \cdot X_2 \cdot X_3 = Y_2 \cdot Y_3 \wedge 2 * \texttt{len}(Z) = \texttt{len}(Y_2) \wedge X_1 = Y_1 \cdot Z \wedge \ldots$$

Similarly, we can continue to discharge the new length constraint in the following step.

In fact, given an input formula, string solvers based on the Makanin's approach usually try to repeatedly apply reduction rules in order to reduce string equations into formulas in solved form or the alike. For example, the splitting rule is used to produce a derivation tree in Figure 2 for an equation

$$Eq \equiv X_1 \cdot X_2 \cdot X_3 = Y_1 \cdot Y_2 \cdot Y_3.$$

Here we omitted the existential variables for simplicity.

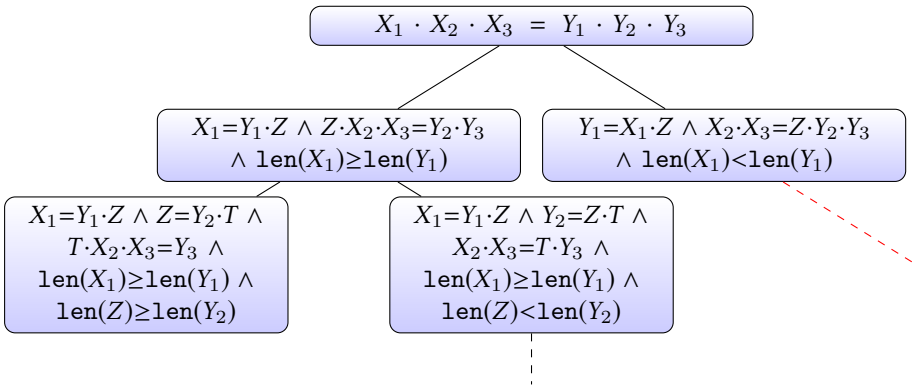

Fig. 2. A derivation tree for a string equation

In Figure 2, by applying the splitting rule to *Eq*, we have the two disjuncts corresponding to the two cases based on the relationship between $\texttt{len}(X_1)$ and $\texttt{len}(Y_1)$. (To create the binary splits, we just fold the $\texttt{len}(X_1) = \texttt{len}(Y_1)$ case into one of the others via non-strict equality.) If we continue to apply the splitting rule to the first disjunct

$$X_1 = Y_1 \cdot Z \wedge Z \cdot X_2 \cdot X_3 = Y_2 \cdot Y_3 \wedge \texttt{len}(X_1) \geq \texttt{len}(Y_1)$$

we will obtain a new disjunctive formula. Now, the two new disjuncts correspond to the two cases based on the relationship between $\texttt{len}(Z)$ and $\texttt{len}(Y_2)$. Similarly, we also have the right-most branches (colored with red) based on the relationship between $\texttt{len}(Z)$ and $\texttt{len}(X_2)$, which are omitted for simplicity. In the first disjunct of the new formula (the left-most node), all equations

$$X_1 = Y_1 \cdot Z \quad , \qquad Z = Y_2 \cdot T \quad , \qquad T \cdot X_2 \cdot X_3 = Y_3$$

are in solved form.

The derivation tree is navigated via the so-called branching decision, which is formalized as below. Basically, an application of a multi-reduct rule will introduce multiple branching choices. For example, the splitting rule as described in Example 2.1 introduces 3 branches, guarded by boolean variables which are to describe the relationship between the lengths of variables (involving the alignments between two sides of a string equation). If there are multiple branching choices we can always make it become *binary-choice* branches by using nested conditions. In fact, we will ensure that the splitting rule will always create two branches (as in Figure 2). We will discuss how we can enforce this later in Section 5.

Definition 4.4 (Branching Decision). *A branching decision is an assignment for a boolean literal, whose value (*`true` *or* `false`*) corresponds to the two branching choices.*

At the high level, the branching decision decides the condition on which path in a derivation tree the search will follow. In the SMT/SAT setting, it is described via a boolean literal as in Section 5. These boolean literals are very important; they will help control the search strategy inside SMT solvers.

For each leaf node of the derivation tree for a string equation, we can compute the number of applications of the splitting rule such that from the input equation, we obtain a formula where all equations are in solved form. Intuitively, this number corresponds to the number of *branching decisions* made (starting from the root node). This measure is formalized as follows. In the below definition, we use the θ function to specify the number of length (and extended) constraints where a variable X_j appear; the greater the number is, the higher priority the variable is of. We will discuss how we use it and its purpose later.

Definition 4.5 (Solved Form Distance). *Let F be a formula and $V = \{X_1, ..., X_n\}$ be a sequence of string variables such that $\texttt{Var}(F) \subseteq V$. Let θ be a mapping from V to the set of integer numbers, e.g. $\theta(X_j) = a_j$.*

The solved form distance $dist(F)_{V,\theta}$ for F wrt. V and θ is a n-tuple defined as follows:

- *If F is* `false` *then the solved form distance $dist(F)_{V,\theta}$ is $(\infty, \ldots, \infty)$.*
- *If F is in solved form wrt. all the variables of F or there is no applicable rule then the solved form distance $dist(F)_{V,\theta}$ is $(b_1, \ldots, b_n)$, where b_j=0 if F is in solved form wrt. X_j, and b_j=∞ otherwise.*

- *Otherwise, we can apply only one derivation rule to F at a time. (This ensures the determinism of our algorithm.) Suppose the derivation rule is* $\frac{F}{\bigvee_{i=1}^{m} G_i}$. *The solved form distance* $dist(F)_{V,\theta}$ *is defined recursively as follows:*

$$dist(F)_{V,\theta} = dist(G_1)_{V,\theta} \qquad \text{if } m = 1$$

$$dist(F)_{V,\theta} = \texttt{my\_min}(\{dist(G_i)_{V,\theta}\}) + (r_1, \ldots, r_n) \qquad \text{if } m > 1$$

where $r_j = 0$ *if* F *is in solved form wrt.* X_j
$r_j = 1$ *otherwise*

□

In our implementation, we choose the set of variables V as the whole set of string variables of the *input* formula and use Skolem functions to avoid introducing new variables when applying rules (as shown in Section 5). As such, we will always use the same set V.

The $\texttt{my\_min}(\{dist(G_i)_{V,\theta}\})$ function is used to get the minimum tuple among all the distances $dist(G_i)_{V,\theta}$ based on the COMPARE function as defined in Pseudocode 1. The COMPARE function takes as its input two solved-form distances d_1, d_2 and a list α. Then it returns the shorter distance between d_1 and d_2 based on α.

```
function COMPARE(d1, d2: solved form distances, α: a list of list of integers)
⟨1⟩   foreach element β in α do                          /* β is a list of integers */
⟨2⟩       foreach element k in β do
⟨3⟩           if d1[k] < d2[k]
⟨4⟩               return d1
⟨5⟩           if d1[k] > d2[k]
⟨6⟩               return d2
⟨7⟩   return d1
```

Pseudocode 1: The function to compare two distances

We first explain how we obtain the list α from θ. Each element of α is a list β of indexes of variables, which appear in the same number of length (and extended) constraints, in V. In other words, there exists some natural number p such that $\forall k \in \beta : \theta(X_k) = p$. We also call p a priority value since the greater p is, the higher priority the variable (at the index k) is of; we then later prioritize achieving solved form formulas wrt. the variable of higher priority. Furthermore, the list α is sorted such that the priority value is decreased. Meanwhile the list β is in an ascending order. In short, the list α is obtained from the mapping function θ by grouping the indexes of variables (e.g. in V) that share the same priority value (e.g. p). Here, we just want to prioritize the variable that appears in the greatest number of length constraints and extended string constraints. To illustrate, let us look at the following formula.

Example 4.6 (String Equations and Complicated Length Constraints).

$$F \equiv X_1 \cdot X_2 \cdot X_3 = Y_1 \cdot Y_2 \cdot Y_3 \wedge \texttt{len}(X_2) = 2 * \texttt{len}(Y_2) \wedge \texttt{len}(X_1) + \texttt{len}(X_2) = 3 * \texttt{len}(Y_1)$$

In addition to the equation Eq as in Figure 2, here we also have two length constraints. According to Definition 4.5, we have $\theta(X_1) = 1, \theta(Y_1) = 1, \theta(X_2) = 2, \theta(Y_2) = 1, \theta(X_3) = 0, \theta(Y_3) = 0$ because X_2 appears in two length constraints, X_1, Y_1, and Y_2 appear in one length constraint, X_3 and Y_3 do not appear in any length constraint. In short, given a sequence V=$\{X_1, Y_1, X_2, Y_2, X_3, Y_3\}$, then θ is

a map from V to $\{1, 1, 2, 1, 0, 0\}$. So we have $\alpha = [[2], [0, 1, 3], [4, 5]]$ since the variable at the index 2 is X_2, which appears in the greatest number of length constraints. (Following it are the variables X_1, Y_1, Y_2 at indexes $0, 1, 3$ and lastly the variables X_3, Y_3 at indexes $4, 5$.) The purpose here is to prioritize finding the value for X_2, which will help us either find a satisfying assignment for X_2 quicker or learn a better conflict clause among constraints of the two theories. □

As we can see in Pseudocode 1, if any element (e.g. at the index k) of the tuple d_1 is less than the corresponding one of d_2 then d_1 is shorter. We refer to such k as the index that decides the minimum tuple. This ensures that d_1 helps us reach *faster* the solved form formula where *more* length (and extended) constraints can be discharged.

Continuing with Example 4.6, according to Def. 4.5 and the tree in Figure 2, we have

$$dist(X_1 \cdot X_2 \cdot X_3 = Y_1 \cdot Y_2 \cdot Y_3)_{V,\theta} = (1, 1, 2, 2, 2, 2).$$

There are two paths that lead to this solved form distance. Each path is encoded as a list of branching decisions that will later be used to guide the search. Given

$$e_1 \equiv \texttt{len}(X_1) \geq \texttt{len}(Y_1) \qquad e_2 \equiv \texttt{len}(Z) \geq \texttt{len}(Y_2) \qquad e_3 \equiv \texttt{len}(Z) \geq \texttt{len}(X_2)$$

the two paths to *witness* the above distance are encoded as

$$l_1 = [e_1 \leftarrow \texttt{true}, e_2 \leftarrow \texttt{true}] \text{ and } l_2 = [e_1 \leftarrow \texttt{false}, e_3 \leftarrow \texttt{true}].$$

The first one corresponds to the left most branch in Figure 2. These lists will be the witnesses for us to replay the solving process later in the global context.

Definition 4.7 (Witness). *A witness for a solved form distance $dist(F)_{V,\theta}$ is a list of branching decisions such that if we follow these branching decisions in the derivation tree for F then the resulting formula is in solved form wrt. all the variables of F.* □

When there are multiple witnesses (e.g. l_1, l_2 in the above example) for one solved form distance, either of them is chosen to guide the search. Our implementation prioritizes inequality than equality (and non-strict inequality than strict inequality) conditions. As such, for this example, l_1 will be chosen.

Now, we can define a partial order for string equations by using the distance from a string equation to its corresponding solved form. So the equations will be solved following the order, from left to right. Let us define the function $\texttt{my\_leq}(d_1, d_2)$ to be $(\texttt{my\_min}(\{d_1, d_2\}) \equiv d_1)$. In other words, `my_leq` returns a boolean value instead of the minimum value between d_1 and d_2 as in `my_min`.

Definition 4.8 (Partial Order for String Equations). *Given two string equations F, G and a sequence of string variables V such that $\texttt{Var}(F) \cup \texttt{Var}(G) \subseteq V$, and a mapping θ from the set V to the set of integer numbers, we have*

$$F \preceq_{V,\theta} G \stackrel{def}{=} \texttt{my\_leq}(dist(F)_{V,\theta}, dist(G)_{V,\theta})$$ □

Let us use Ex. 2.3 to demonstrate all the above definitions. We need to solve the formula:

$$F \equiv Eq_1 \wedge Eq_2 \wedge H$$

where $Eq_1 \equiv Y_1 \cdot Z_2 = \text{“}a\text{”} \cdot Y_2$ and $Eq_2 \equiv X_1 \cdot Y_1 \cdot Z_1 = X_2 \cdot \text{“}a\text{”} \cdot Z_2$ and $H \equiv \neg\texttt{contains}(Z_2, \text{“}a\text{”})$. To proceed, we need to choose which equation to apply the splitting rule to first. Suppose we have a sequence of variables $V = \{X_1, X_2, Y_1, Y_2, Z_1, Z_2\}$. Then θ is $(0, 0, 0, 0, 0, 1)$ since we have an extended string constraint on Z_2. We proceed by computing $dist(Eq_1)_{V,\theta}$ and $dist(Eq_2)_{V,\theta}$ as in Figure 3 and Figure 4 respectively. By Definition 4.5, we have

$$d_1 = dist(Eq_1)_{V,\theta} = (\infty, \infty, 1, 1, \infty, 1) \qquad d_2 = dist(Eq_2)_{V,\theta} = (1, 1, 2, \infty, 2, 2)$$

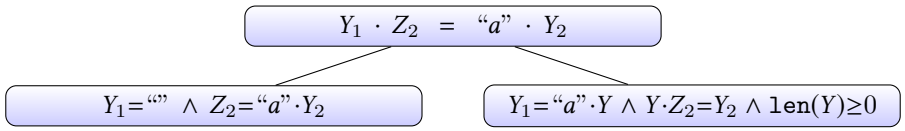

Fig. 3. A derivation tree for Eq_1

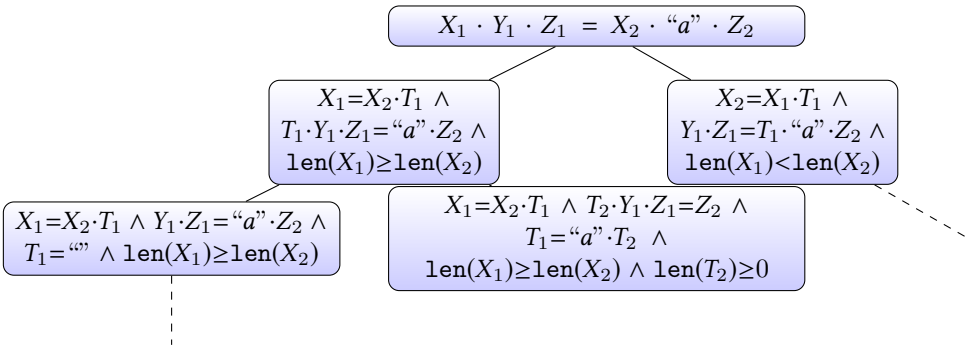

Fig. 4. A derivation tree for Eq_2

Because $\texttt{my\_min}(\{d_1, d_2\}) \equiv d_1$, we will apply the splitting rule to Eq_1 first.

When computing d_1, we know not only which equation to be processed, but also which search path to proceed. This is because we have collected the witness for the distance d_1 as in Figure 3. Because we have the same solved-form distance for both paths, we may use either of them as the witness for d_1. These are $l_1 = [e_1 \leftarrow \texttt{true}]$ and $l_1' = [e_1 \leftarrow \texttt{false}]$ where $e_1 \equiv len(Y_1){=}0$.

Though we will follow l_1' in our implementation, we will discuss both choices here. Now, in the solving process, Eq_1 will be processed first. The splitting rule will be in charge of breaking Eq_1 into two disjuncts

$$G_1 \equiv Y_1 = \text{“”} \wedge Z_2 = \text{“}a\text{”} \cdot Y_2 \qquad G_2 \equiv Y_1 = \text{“}a\text{”} \cdot Y \wedge Y \cdot Z_2 = Y_2 \wedge \texttt{len}(Y) \geq 0$$

Importantly, which one of them will be processed next is guarded by the boolean literal e_1. If we choose to follow the right path as in our implementation (ie. $e_1 \equiv \texttt{false}$), the new formula will be

$$G_2 \wedge Eq_2 \wedge H$$

Therefore, we can easily find a satisfying assignment for every string variable without any backtracking. For example, if Y is “” then one solution is that Y_1 is “a” and X_1, Z_1, X_2, Y_2, Z_2 are “”.

Even if we follow the left path (i.e. $e_1 \equiv \texttt{true}$), the cost of backtracking is still much less expensive than working on the equation Eq_2. In this case, the new formula will be $G_1 \wedge Eq_2 \wedge H$. Because a conflict is detected between $Z_2 = \text{“}a\text{”} \cdot Y_2$ and $\neg\texttt{contains}(Z_2, \text{“}a\text{”})$, we have to backtrack with a new axiom that is $Y_1 \neq \text{“”}$. Next, by re-analysing the “new" equation Eq_1, whose derivation tree now contains only *one* path, we can obtain the same result as above.

Theorem 4.9 (Soundness). *Our dependency analysis is sound.* □

Basically, our analysis does *not* affect the soundness of the base solver since it only affects the order in which string constraints are reduced and the order of which branch to follow. In other words, it does not lose any solution or add any incorrect solution to the base solving algorithm. Thus, if the base solving algorithm is sound then our whole algorithm is sound.

5 DPLL(T) STRING SOLVER

In this section, we present the implementation of our inter-theory dependency analysis inside an SMT solver. We also highlight the technical challenges that we have to deal with. Though we implemented our dependency analysis in the string solver of Z3, the implementation can be adapted to work with other SMT solvers as well.

Before describing how our string solver works as part of an SMT solver, we briefly recall the architecture of an SMT solver such as Z3. The core component consists of the following modules: the congruence closure engine, a SAT solver-based DPLL layer, and several built-in theory solvers such as integer linear arithmetic, bit-vectors, etc. The congruence closure engine can detect equivalent terms and then classify them into different equivalence classes, which are shared among all built-in theory solvers. The SAT-based DPLL layer is responsible for handling the boolean structure of the input formula.

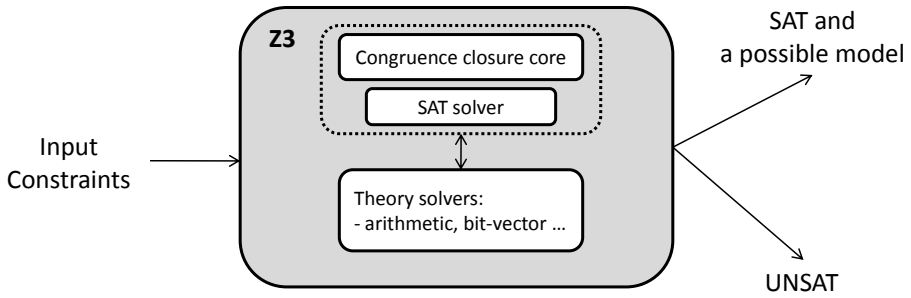

Fig. 5. The Architecture of Z3

As an SMT solver, Z3 contains multiple built-in theory solvers to handle different types of constraints. The string solver is in charge of handling string equations and extended string constraints, but because of length constraints, there will be interaction between string and integer solvers. For example, our string solver may query about the relationship between the lengths of string variables from the integer solver and also propagates more length constraints to the integer solver. As mentioned above, one main technical challenge for that comes from the separation between the two theory solvers: string and integer solvers. Since the only thing that is shared among different theory solvers is the congruence closure core, we will try to query from that the length relations in order to solving string constraints as in Section 5.2. However, this is still very limited and mostly based on the syntax of length constraints. As such, a systematic method to deal with length (and extended string) constraints is based on our dependency analysis as in Section 5.3.

Let us use Example 2.2 to illustrate how Z3 works. First, the Z3 core component treats the three equations as three independent boolean variables e_1, e_2, e_3 such that

- $e_1 \equiv X \cdot \text{“}a\text{”} \cdot Y = Z \cdot T$,
- $e_2 \equiv X \cdot \text{“}a\text{”} \cdot Y = X_1 \cdot \text{“}b\text{”} \cdot Y_1$, and
- $e_3 \equiv \texttt{len}(X) = \texttt{len}(X_1)$.

Then it tries to assign boolean values, either `true` or `false`, to them. Of course, in this case, the `true` value is the only option. W.r.t. to this value, constraints will be distributed to corresponding theory solvers. For example, the two equations are sent to the string solver, while the length constraint is sent to integer solver. When encountering a new equation, each solver will merge the equivalence classes of the two sides of the equation and update the congruence closure core. So the closure core will have two equivalence classes:

- the first one contains 3 terms: $X \cdot$ "a" $\cdot\, Y$ and $Z \cdot T$ and $X_1 \cdot$ "b" $\cdot\, Y_1$.
- the second contains 2 terms: $\texttt{len}(X)$ and $\texttt{len}(X_1)$.

All theory solvers will share the congruence closure core. In each theory solver, we need to provide methods to solve these newly-added constraints.

Pseudocode 2 presents the main function SOLVE of the string theory solver to handle string equations and extended string constraints. This function takes as input a list (i.e., conjunction) of string equations *eqs* and a list of extended string constraints *exts*. Here, we omitted the reasoning about a list of string disequations and the handling of extended string terms such as `replace` function for simplicity. These can be done similarly to previous works such as [Reynolds et al. 2017; Trinh et al. 2016]. Since we collect the list *eqs* of string equations, we can completely control the order in which string equations are being processed. The order is expressed via our priority queue in Section 5.3. We can also control the order in which disjunctive formulas introduced by the splitting rule are being processed. This is done via the boolean value assigned to the branching variable; we will discuss this in Section 5.1.

```
function SOLVE(eqs: a list of equations, exts: a list of extended constraints)
⟨1⟩  if (SIMPLIFY_EQS(eqs))                    /* apply simplification rules to equations */
       return CONTINUE
⟨2⟩  if (FIXED_LENS())                  /* propagation for variables whose lengths are fixed*/
       return CONTINUE
⟨3⟩  if (SPLIT_W_LENS(eqs))                    /* split based on length information */
       return CONTINUE
⟨4⟩  if (SPLIT_WO_LENS(eqs))                   /* split based on dependency analysis */
       return CONTINUE
⟨5⟩  if (CHECK_LEN_CONSISTENCY())                      /* check length consistency */
       return CONTINUE
⟨6⟩  if (CHECK_EXT_CONTS(exts))          /* check if all extended constraints are satisfied */
       return CONTINUE
⟨7⟩  if (CHECK_IS_SOLVED())              /* check if all string variables have been solved */
       return DONE
     return GIVEUP
```

Pseudocode 2: The main function to handle string constraints

In Pseudocode 2, there are seven auxiliary functions. The first one is to apply simplification rules, described in the next subsection, to string equations. The second one is to propagate the values of variables whose length is fixed. The third and fourth ones are to apply instances of the splitting rules, also described in the next subsection, to string equations. The last three are to check if there is any inconsistency among length constraints, if there is any inconsistency among extended string constraints and if every string variable is assigned with some value respectively. If CHECK_LEN_CONSISTENCY() returns `false` (i.e. there is a conflict), the solver will backtrack. If CHECK_IS_SOLVED() returns `false`, it means that the solver cannot find satisfying assignments for all

string variables though the formula is consistent. In this case, the solver will return the UNKNOWN answer.

Function SOLVE is called by the Z3 core as a final step in its try-and-backtrack process. There are three possible returning results: CONTINUE, GIVEUP, and DONE. (Similar things would be done for other theory solvers.)

- If the function returns CONTINUE, then the core will know that there is a new propagation. So it checks the consistency of the current formula. If the formula is consistent, it continues the search. Otherwise, it backtracks.
- If it returns GIVEUP, the core will continue with propagation steps in other theories.
- If it returns DONE, the core will continue with propagation steps in other theories.

At the top level (Z3 core), when no more propagation is possible then:

- if all the theory solvers return DONE then the search terminates with SAT answer;
- if one of the theory solvers return GIVEUP then the search terminates and Z3 returns UNKNOWN as the answer; and
- if the search is already exhausted and we find conflicts in all the branches of the tree then Z3 will return UNSAT.

In Pseudocode 2, our contributions are on SPLIT_W_LENS and SPLIT_WO_LENS, where the splitting rules is used. In the following, we first present different instances of the splitting rule governing the very *bare* version of SPLIT_WO_LENS. Then we discuss how we can improve it and overcome the above technical challenge (of theory separation) by making use of i) length offset information from the congruence closure engine to support splitting with lengths in the SPLIT_W_LENS function, and ii) inter-theory dependency analysis to support smarter splitting in the SPLIT_WO_LENS function.

5.1 The Splitting Rule

We first explain how the splitting rule is implemented in an SMT solver. Then we present a list of instances of the splitting rule. Given an equation $X \cdot Y = Z \cdot T$, SPLIT_WO_LENS introduces a boolean variable e to represent $\texttt{len}(X) \geq \texttt{len}(Z)$. Based on the boolean value assigned to e by the SAT solver, we may proceed with one of two cases. Here, we can also force the value of a boolean literal to be `true` or `false` (e.g. using the `force_phase()` function in Z3). Thus we are able to control the order in which the next formula will be processed.

- If e is `false` then SPLIT_WO_LENS propagates $\exists X_1 : Z{=}X{\cdot}X_1 \wedge Y{=}X_1{\cdot}T \wedge \texttt{len}(X_1){>}0$
- If e is `true` then SPLIT_WO_LENS propagates $\exists Z_1 : X{=}Z{\cdot}Z_1 \wedge Z_1{\cdot}Y{=}T \wedge \texttt{len}(Z_1){\geq}0$

For the inferred equations such as $Z = X \cdot X_1$, they will be handled similarly to the equations in the input formula. In our implementation, we do not introduce existential variables. Instead, we use the Skolem functions to represent them. For example, we use $seq.right(Z, X)$ to represent X_1. We can formalize the propagation step as the following splitting rule:

$$\frac{X \cdot Y = Z \cdot T}{\exists X_1 : (Z = X \cdot X_1 \wedge Y = X_1 \cdot T \wedge \texttt{len}(X_1){>}0) \ \vee \ \exists Z_1 : (X = Z \cdot Z_1 \wedge Z_1 \cdot Y = T \wedge \texttt{len}(Z_1){\geq}0)}$$

Before presenting our instances of the splitting rule, let us introduce the classification of string terms based on their syntax. Each side of a string equation is of one of the following five forms:

(1) a
(2) $a_1 \cdot E \cdot a_2$, where E is not a constant string
(3) $X \cdot E \cdot a$ or $a \cdot E \cdot X$
(4) $X \cdot D \cdot a \cdot E \cdot Y$
(5) $X_1 \cdots X_n$

The first form indicates a constant string. The second one indicates an expression where both head and tail are constant but the whole expression is not. The third one indicates an expression with either head or tail is constant (but not both). The fourth one indicates an expression with both head and tail are variables but there is still some constant string inside the expression. We use D, E to represent string expressions (which might contain string constants) while X, Y are to represent string variables. We define these notations in Section 3. So $X \cdot D \cdot a \cdot E \cdot Y$ is more general than $X \cdot a \cdot Y$. The meaning of $X \cdot D \cdot a \cdot E \cdot Y$ is that i) it starts and ends with variables ii) it contains *at least* one string constant, that is a. The last one is the concatenation of string variables. Note that the classification is done after our unification step, where we substitute every string variable in a side of an equation with its most grounded term. For example, if we have $X_1 \cdot Y_1 = X_2 \cdot Y_2 \wedge X_1 =$ "c" $\wedge\ Y_2 = Z_1 \cdot Z_2$ then the unification step will rewrite the first equation into "c" $\cdot\ Y_1 = X_2 \cdot Z_1 \cdot Z_2$. Thus, the LHS of this equation is of type (3) and the RHS is of type (5).

$$(3,3)\quad \frac{a \cdot X_2 \cdot D_2 \cdot Y_2 = Z_1 \cdot E_1 \cdot T_1 \cdot b}{\bigvee_{i=0}^{|a|}(a_0^i{=}Z_1{\cdot}E_1{\cdot}T_1 \wedge a_i^{|a|}{\cdot}X_2{\cdot}D_2{\cdot}Y_2{=}b)\ \vee\ (\exists X_1 : a{\cdot}X_1{=}Z_1{\cdot}E_1{\cdot}T_1 \wedge X_2{\cdot}D_2{\cdot}Y_2{=}X_1{\cdot}b)}$$

$$(2,4)\quad \frac{a \cdot X_2 \cdot D_2 \cdot Y_2 \cdot c = Z_1 \cdot E_1 \cdot T_1 \cdot b \cdot Z_2 \cdot E_2 \cdot T_2}{\begin{array}{l}\bigvee_{i=0}^{|a|}\bigvee_{j=0}^{|c|}(a_0^i = Z_1 \cdot E_1 \cdot T_1 \wedge a_i^{|a|} \cdot X_2 \cdot D_2 \cdot Y_2 \cdot c_0^j = b \wedge c_j^{|c|} = Z_2 \cdot E_2 \cdot T_2) \vee \\ \bigvee_{j=0}^{|c|}(\exists X_1 : a \cdot X_1 = Z_1 \cdot E_1 \cdot T_1 \wedge X_2 \cdot D_2 \cdot Y_2 \cdot c_0^j = X_1 \cdot b \wedge c_j^{|c|} = Z_2 \cdot E_2 \cdot T_2) \vee \\ \bigvee_{i=0}^{|a|}(\exists Y_1 : a_0^i = Z_1 \cdot E_1 \cdot T_1 \wedge a_i^{|a|} \cdot X_2 \cdot D_2 \cdot Y_2 = b \cdot Y_1 \wedge Y_1 \cdot c = Z_2 \cdot E_2 \cdot T_2) \vee \\ (\exists X_1, Y_1 : a \cdot X_1 = Z_1 \cdot E_1 \cdot T_1 \wedge X_2 \cdot D_2 \cdot Y_2 = X_1 \cdot b \cdot Y_1 \wedge Y_1 \cdot c = Z_2 \cdot E_2 \cdot T_2)\end{array}}$$

$$(3,4)\quad \frac{a \cdot X_2 \cdot D_2 \cdot Y_2 = Z_1 \cdot E_1 \cdot T_1 \cdot b \cdot Z_2 \cdot E_2 \cdot T_2}{\begin{array}{l}\bigvee_{i=0}^{|a|}(a_0^i = Z_1 \cdot E_1 \cdot T_1 \wedge a_i^{|a|} \cdot X_2 \cdot D_2 \cdot Y_2 = b \cdot Z_2 \cdot E_2 \cdot T_2) \vee \\ (\exists X_1 : a \cdot X_1 = Z_1 \cdot E_1 \cdot T_1 \wedge X_2 \cdot D_2 \cdot Y_2 = X_1 \cdot b \cdot Z_2 \cdot E_2 \cdot T_2)\end{array}}$$

$$(3,4)\quad \frac{X_2 \cdot D_2 \cdot Y_2 \cdot c = Z_1 \cdot E_1 \cdot T_1 \cdot b \cdot Z_2 \cdot E_2 \cdot T_2}{\begin{array}{l}\bigvee_{j=0}^{|c|}(c_j^{|c|} = Z_2 \cdot E_2 \cdot T_2 \wedge X_2 \cdot D_2 \cdot Y_2 \cdot c_0^j = Z_1 \cdot E_1 \cdot T_1 \cdot b) \vee \\ (\exists Y_1 : Y_1 \cdot c = Z_2 \cdot E_2 \cdot T_2 \wedge X_2 \cdot D_2 \cdot Y_2 = Z_1 \cdot E_1 \cdot T_1 \cdot b \cdot Y_1)\end{array}}$$

$$(4,4)\quad \frac{X_1 \cdot D_1 \cdot Y_1 \cdot a \cdot X_2 \cdot D_2 \cdot Y_2 = Z_1 \cdot E_1 \cdot T_1 \cdot b \cdot Z_2 \cdot E_2 \cdot T_2}{\begin{array}{l}(\exists X_3 : X_1 \cdot D_1 \cdot Y_1 = Z_1 \cdot E_1 \cdot T_1 \cdot X_3 \wedge X_3 \cdot a \cdot X_2 \cdot D_2 \cdot Y_2 = b \cdot Z_2 \cdot E_2 \cdot T_2) \vee \\ (\exists X_4 : X_1 \cdot D_1 \cdot Y_1 \cdot X_4 = Z_1 \cdot E_1 \cdot T_1 \wedge a \cdot X_2 \cdot D_2 \cdot Y_2 = X_4 \cdot b \cdot Z_2 \cdot E_2 \cdot T_2)\end{array}}$$

Fig. 6. Some Instances of The Splitting Rule

Based on the syntax of two sides of an equation, we can have different instances of the splitting rule. To illustrate, we list some of them in Figure 6. If we use a pair of integer numbers (i, j) to represent the kind of the syntax of the LHS and RHS of an equation, we can map the rule instances in Figure 6 to (3, 3), (2, 4), (3, 4), (3, 4), (4, 4). Note that although we use a slightly different syntax in Figure 6, they represent the same forms of expressions as shown above. For example, $a \cdot X_2 \cdot D_2 \cdot Y_2$ is of type (3) since we can map E to $X_2 \cdot D_2$ and X to Y_2. The purpose of using the variable X_2 is to state that a is the longest string constant. The notation a_i^j denotes the substring of a from the bound i to the bound j. Let us summarize all the instances of our splitting rule in the left-right order: (3, 3), (2, 4), (3, 4) (4, 4), (1, 4), (1, 5), (2, 5), (3, 5), (4, 5), (5, 5).

Other Rules. There are two kinds of derivation rules: one-reduct rules and multi-reduct rules. An example of multi-reduct rules is the splitting rule while an example of one-reduct rules is the simplification rule. For reference, we list some important one-reduct rules governing step 1 in Pseudocode 2. Below are simplification rules based on the syntax above:

- 1-real-var: $X = a$
- $(1, 1)$ is simplified into `true` or `false`
- $(1, 2)$ is simplified into $(1, 4)$ or $(1, 5)$
- $(1, 3)$ is simplified into $(1, 4)$ or $(1, 5)$
- $(2, 3)$ is simplified into $(2, 4)$ or $(2, 5)$ or $(3, 3)$ or $(3, 4)$ or $(3, 5)$
- $(2, 2)$ is simplified into $(2, 3)$

Other kinds of simplification rules are listed in Figure 7. The (SUB) rule is to substitute all variables X in F with E, where E is some term/expression. We need to keep track of the previous substitution to avoid going into a loop of application of this rule. The (CON) rule is to detect the contradiction among string constants.

$$\text{(CON)}\quad \frac{F \wedge a = b}{\texttt{false}} \qquad a, b \text{ are constant strings and } a \neq b$$

$$\text{(SUB)}\quad \frac{F \wedge X = E}{F[E/X] \wedge X = E} \qquad X \in \texttt{Var}(F) \text{ and } E \text{ is a string term}$$

Fig. 7. Simplification Rules for String Constraints

The order in Figure 6 indicates the priority between the instances of the splitting rule. However, such order is only used to break a tie after the two following orders have been used.

5.2 Splitting with Length Offsets

Every equation interacting with length constraints will be solved first via SPLIT_W_LENS in Pseudocode 2. Function SPLIT_W_LENS (in step 3) will apply specialized instances of the splitting rule. In our implementation, SPLIT_W_LENS processes the splitting based on the offset relationship between lengths of variables. For example, if we have

$$D_1 \cdot D_2 = E_1 \cdot E_2 \wedge \texttt{len}(D_1) = \texttt{len}(E_1) + \texttt{offset},$$

we can reduce it into

$$D_1 = E_1 \cdot X \wedge X \cdot D_2 = E_2 \wedge \texttt{len}(X) = \texttt{offset}.$$

Here, `offset` can also be a variable. Note that, these rules are mostly based on the offset between lengths of string variables. A systematic way to deal with length constraints is still based on the partial order of string equations, which will be presented later.

Also, we transform equations in order to utilize as much as possible length constraints. Specifically, based on the equivalence classes of the LHS and RHS of an equation, we are able to choose the corresponding equivalent terms (for the two sides) such that the new equation will resolve such length constraints. To illustrate, let us look at the following example.

Example 5.1 (Equation Rewriting and Splitting Guided by Length Constraints).

$$X \cdot \text{“}a\text{”} \cdot Y = Z \cdot T \wedge Z \cdot T = X_1 \cdot \text{“}b\text{”} \cdot Y_1 \wedge \texttt{len}(X) = \texttt{len}(X_1) - 1$$

In this example, there is a relationship between the length of X and the length of X_1, yet X and X_1 do not appear in a same string equation. A naive algorithm may choose to apply the splitting rule to an arbitrary equation, leading to a lot of backtracking.

In contrast, making use of the length constraint $\texttt{len}(X) = \texttt{len}(X_1) - 1$, our algorithm instead transforms the input formula into:

$$X \cdot \text{“}a\text{”} \cdot Y = X_1 \cdot \text{“}b\text{”} \cdot Y_1 \wedge Z \cdot T = X_1 \cdot \text{“}b\text{”} \cdot Y_1 \wedge \texttt{len}(X) = \texttt{len}(X_1){-}1$$

Similarly to Example 2.2, we now can apply the splitting rule to the first equation to obtain only *one* possible subcase:

$$X_1 = X \cdot X_2 \wedge \texttt{len}(X_2) = 1 \wedge \text{“}a\text{”} \cdot Y = X_2 \cdot \text{“}b\text{”} \cdot Y_1 \wedge Z \cdot T = X_1 \cdot \text{“}b\text{”} \cdot Y_1$$

Next, because $\texttt{len}(X_2) = 1$, we have:

$$X_1 = X \cdot X_2 \wedge X_2 = \text{“}a\text{”} \wedge Y = \text{“}b\text{”} \cdot Y_1 \wedge Z \cdot T = X_1 \cdot \text{“}b\text{”} \cdot Y_1$$

We now can easily derive assignments for all variables *without* any backtracking. □

5.3 Splitting with Dependency Analysis

Function SPLIT_WO_LENS (in step 4) will apply instances of the splitting rule in Section 5.1. To make use of the result of our dependency analysis, we implement the partial order for string equations as a *priority queue*. In other words, what we have is that *eqs* is a priority queue instead of a list as in Pseudocode 2. To do so, we associate the priority of each equation with its solved form distance w.r.t. a sequence of variables V and the function θ. We also collect the list of branching decisions as in Section 4 when computing the solved form distance for each string equation in the priority queue. The implementation basically follows the algorithm presented in the previous section though there are a few things to note as below.

As we consider the original variable set V and do not introduce new variables, all equations will be compared wrt. the same set of variables. The θ function is recomputed whenever the detected conflict is due to some length (or extended) constraint.

To be efficient when computing the solved form distance for an equation, instead of exploring the whole derivation tree, whose depth may be too large, we set the maximum depth to be $n + 1$, where n is the value at the index that decides the current minimum distance tuple. In fact, for the benchmarks in our experimental evaluation, n is usually 1 or 2. This is because with the help of the function SPLIT_W_LENS, we usually can make use of length constraints to simplify current equations to their simpler forms.

As also mentioned previously in Section 4, since we only explore up to a certain depth of a local derivation tree (when computing the solved form distance for an equation), we can avoid the case of (local) infinite trees, for example, caused by overlapping variables. But since this is only a local attempt, we can still encounter an infinite tree caused by overlapping variables in the global context.

Surprisingly, our technique does in fact mitigate such non-termination problem in the global context. Specifically, this also helps us find a solution in many cases where the formula contains overlapping variables. This is demonstrated in our experiments with benchmarks generated by the Kudzu framework. Specifically, our new solver can decide the satisfiability of the benchmarks that Z3-str3 detects overlapping variables. The reason is that according to our inter-theory dependency analysis, in those benchmarks, the variables of interest (i.e. those appearing in the largest number of length constraints) are usually not overlapping variables. Thus, the solver will prioritize finding solutions for those (non-overlapping) variables. These solutions enable the reduction of the string equations on the overlapping variables to simpler forms such as having fixed lengths or having

trivial conflicts, which in turn helps avoid applying the splitting rule to the string equations involving overlapping variables and ensure the termination of our solver.

6 EVALUATION

We implemented our dependency analysis on top of the sequence theory solver of Z3. Our solver is called S3N, where N stands for the native S3. We evaluated our solver against the state-of-the-art, using two case studies which contain practical constraints generated from testing JavaScript and Python applications. These benchmark suites are also used in the evaluation of the state-of-the-art solvers. We also use the same timeout, 20 seconds, as in previous works.

6.1 Experiments with JavaScript Benchmarks

In the first case study, we used a large and popular set of benchmark constraints generated using Kudzu [Saxena et al. 2010], a symbolic execution framework for JavaScript. This set contains a lot of string equations and length constraints but no extended string constraints such as `replace` operations. Note that the constraints in these benchmarks have already been preprocessed and/or over-simplified. In particular, the string lengths have been bounded and recursive string function such as `replace` have been transformed to primitive operators so that the underlying solver of Kudzu can handle.

Table 1 summarizes the results of running state-of-the-art solvers including CVC4 (v-1.5 [Reynolds et al. 2017] and v-1.7 [Reynolds et al. 2019]), S3P [Trinh et al. 2016], Z3str3 [Berzish et al. 2017] and Z3. We also compare our solver with its bare version where the dependency analysis is removed. We call it $Z3_b$ since it is basically Z3 facilitated with more instances of the splitting rule, which are adopted from state-of-the-art solvers. Since S3N is developed on top of Z3 commit `082936b`, we use such version of all Z3-based tools for comparison.

In Table 1, the first block contains all benchmarks that at least one solver produced a correct model for. For this category, we also cross-verify the models produced by solvers. The second block contains the benchmarks where CVC4(s), S3P, and S3N produced an UNSAT answer. In rows `Total(s)` and `Time(s)`, we report for each tool the total running time and the aggregated running time but only for the correctly solved instances respectively. This is because the total running time alone can be easily distorted by choosing a different timeout. We also report the speedup of S3N for the two cases. These are denoted as $\texttt{speedup}_{total}$ and speedup respectively.

Table 1. Experiments with Kudzu benchmarks.

		Z3	$Z3_b$	Z3-str3	S3P	CVC4-1.5	CVC4-1.7	S3N
SAT	Sat	33585	33450	34970	35270	35235	35235	35270
	Unknown	0	0	84	0	0	0	0
	Timeout	1685	1820	216	0	35	35	0
`Total(s)`		34540	37497	7823	6870	6991	9175	828
$\texttt{speedup}_{total}$		41.7	45.3	9.4	**8.3**	**8.4**	**11.1**	
`Time(s)`		840	1097	3500	6870	6291	8475	828
speedup		1.5	1.7	5.8	**8.3**	**8.0**	**10.2**	
UNSAT	Unsat	11799	11799	11799	12014	12014	12014	12014
	Unknown	0	0	215	0	0	0	0
	Timeout	215	215	0	0	0	0	0
`Total(s)`		4618	4738	242	820	888	1512	260
$\texttt{speedup}_{total}$		17.8	18.2	0.9	**3.2**	**3.4**	**5.8**	
`Time(s)`		318	438	238	820	888	1512	260
speedup		1.3	1.7	1	**3.2**	**3.4**	**5.8**	

According to Table 1, CVC4(s), S3P, and S3N are the solvers that produced most correct answers in both SAT and UNSAT categories. Now let us zoom in on the SAT category and summarize with a few remarks, noting that when a solver cannot produce a correct answer then it does not make sense to discuss the speedup:

- Comparing with solvers producing correct answers for roughly similar number of benchmarks, namely CVC4(s) and S3P, our solver S3N is at least 8 times faster.
- Z3 timeouts on a large number (1685) of benchmarks. However, for those it can solve correctly, it is very efficient and the speedup of S3N over Z3 on these cases is only 1.5 times. (In fact, the speedup of S3N over Z3 will become bigger if we increase the timeout. Specifically, Z3 can give the SAT answer to 30 more benchmarks if we set the timeout to be 30s.) A closer investigation reveals that the sequence theory of Z3 is not equipped with complicated instances of the splitting rule. Instead, its last resort is an instance of the splitting rule augmented for some specific patterns on possible length alignments.
- Meanwhile, $Z3_b$ is equipped with more instances of the splitting rule. These instances are also used by the state-of-the-art string solvers. In most of the cases where there is a solution, $Z3_b$ seems to be less efficient than Z3. In the other case—if there is no solution—$Z3_b$ is still useful. We will discuss this point later.
- Z3str3 has a feature to detect patterns of overlapping variables, for which naively performing splitting will lead to an infinite loop. As a result, it quickly returns Unknown answers for such 84 benchmarks.

For the UNSAT benchmarks, the speedup of S3N over the compared solvers is not as marked. While S3N remains much faster (at least 3 times) than CVC4(s) and S3P, which can decide the unsatisfiability for all the benchmarks, it is comparable with Z3str3 *in case* Z3str3 can solve the example at hand. However, S3N still solves more benchmarks than Z3str3.

For this case study, compared to our baseline solvers Z3 and $Z3_b$, our new solver S3N has demonstrated the capability of handling length constraints (along with string equations). At first, adding more instances of the splitting rule [2] makes $Z3_b$ worse than Z3. This is because the search space that has been explored before the last resort method of $Z3_b$ is called becomes bigger. It is demonstrated via the additional 135 benchmarks that $Z3_b$ timeouts (compared to Z3). However, since our new solver S3N also makes use of length constraints to skip exploring the search space introduced by the new instances of the splitting rule, S3N can still achieve a better performance.

6.2 Experiments with Python Benchmarks

The second case study is to test string solvers with constraints generated by PyEx when it is running on a test suite of 19 functions sampled from 4 popular Python packages: `httplib2`, `pip`, `pymongo`, and `requests` [Reynolds et al. 2017]. There are two benchmark suites that are generated using PyEx with different solvers: CVC4 and Z3. Note that, apart from Kudzu benchmarks, these benchmarks contain a lot of extended string constraints since unlike Kudzu, PyEx does not make any simplification to the constraints.

The results are shown in Tables 2-5; the first two tables are for satisfiable benchmark constraints and the other two are for unsatisfiable ones. Here, we use CVC4(s) with its best configuration (i.e. +fs) as shown in [Reynolds et al. 2017]. We do not include S3P in the comparison since it does not support some string operations in these benchmarks such as `prefixOf`. We use * to mark the benchmarks that a solver returns incorrect models.

According to Tables 2-5, S3N not only has the largest number of definitive answers, but its performance also outperforms that of other solvers. For SAT benchmarks (Table 2 and Table 3), the

[2]Note that these instances are adopted from state-of-the-art solvers.

Table 2. Experiments with satisfiable constraints generated by PyEx with CVC4.

		Z3	$Z3_b$	Z3str3	CVC4-1.5	CVC4-1.7	S3N
SAT	Sat	3398	2114	2233	4241	4271	4271
	Unsat	0	0	26	0	0	0
	Unknown	65	43	1	0	0	0
	Error	0	0	71	0	0	0
	Timeout	810	2116	1942	32	2	2
Time(s)		5088	1254	9364	2355	1692	174
speedup		36.8	14.6	102.9	**13.6**	**9.7**	

Table 3. Experiments with satisfiable constraints generated by PyEx with Z3.

		Z3	$Z3_b$	Z3str3	CVC4-1.5	CVC4-1.7	S3N
SAT	Sat	4414	2884	3620	5699	6278	6952
	Unsat	7	0	19	0	0	0
	Unknown	100	59	1	0	0	0
	Error	0	0	116	0	0	0
	Timeout	2510	4088	3275	1332	753	79
Time(s)		5804	5550	13293	4685	6625	364
speedup		25.1	36.8	70.1	**15.7**	**20.2**	

speed-up is at least about 10 times. This outperformance confirms that our dependency analysis is very helpful. There are two reasons that explain this result. Firstly, many extended string constraints can be reduced into string equations and length constraints—the core fragment where we have demonstrated the applicability and efficiency of our dependency analysis as in the first case study. For example, the `indexOf` and `charAt` operations require the length of the substring from the beginning of the string to the position that matched the condition to be equal to some integer (see Section 3). Secondly, in the case the extended constraints cannot be reduced, we can also make use of these constraints to guide the application of the splitting rule to string equations as shown in Ex. 2.3.

Table 4. Experiments with unsatisfiable constraints generated by PyEx with CVC4.

		Z3	$Z3_b$	Z3str3	CVC4-1.5	CVC4-1.7	S3N
UNSAT	Sat	4*	0	19*	0	0	0
	Unsat	1276	1276	1228	1259	1252	1296
	Unknown	2	0	1	0	0	0
	Error	0	0	46	0	0	0
	Timeout	14	20	1	37	44	0
Time(s)		61	68	108	135	119	27
speedup		**2.3**	**2.6**	4.2	5.2	4.6	

For UNSAT benchmarks (Table 4 and Table 5), the speed-up does vary from about 2 to more than 5 times. More importantly, the improvement that S3N brings is in the *number* of benchmarks that S3N can decide unsatisfiability. In fact, because our dependency analysis also takes into account other constraints such as length and extended string constraints, our solving process will focus

Table 5. Experiments with unsatisfiable constraints generated by PyEx with Z3.

		Z3	$Z3_b$	Z3str3	CVC4-1.5	CVC4-1.7	S3N
UNSAT	Sat	9*	0	27*	0	0	0
	Unsat	1330	1359	1304	1336	1358	1383
	Unknown	1	1	7	0	0	0
	Error	0	0	13	0	0	0
	Timeout	43	23	32	47	25	0
Time(s)		58	98	60	113	185	35
speedup		1.7	**2.9**	1.8	3.3	**5.4**	

more on the equations which can be inconsistent with these constraints. As a result, S3N has a better chance of detecting (inter-theory) inconsistency and proving more unsatisfiable formulas.

Similarly to the first case study, for the SAT benchmarks, $Z3_b$ is worse than Z3 with respect to both the number of correct answers and performance since it has to explore a bigger search space before calling the last resort. However, for the UNSAT benchmarks, $Z3_b$ has more correct answers than Z3 does. This is because the specialized splitting rules in Z3 are not complete and not always efficient. This again confirms having a dependency analysis is a more complete way to follow. The dependency analysis will help us achieve a better search strategy though the search space becomes bigger due to more instances of the splitting rule.

7 RELATED WORK

There is a vast literature on the problem of string solving. Practical methods for solving string equations can loosely be divided into bounded and unbounded methods. Bounded methods (e.g., HAMPI [Ganesh et al. 2011], CFGAnalyzer [Axelsson et al. 2008], and [He et al. 2013]) often assume fixed length string variables, then treat the problem as a normal constraint satisfaction problem (CSP). These methods can be quite efficient in finding satisfying assignments and often can express a wider range of constraints than the unbounded methods. However, as also identified in [Saxena et al. 2010], there is still a big gap in order to apply them to constraints arising from the analysis of web applications since the constraints usually involve *unbounded* strings.

To reason about feasibility of a symbolic execution path from high-level programs, of which string constraints are involved, one approach [Bjørner et al. 2009; Saxena et al. 2010] is to proceed by first enumerating concrete length values, before encoding strings into bit-vectors. In a similar manner, [Redelinghuys et al. 2012] addresses multiple types of constraints for Java PathFinder. Though this approach can handle many operators, it provides limited support for `replace`, requiring the result and arguments to be concrete. Furthermore, it does not handle regular expressions. Importantly, all of them have performance limitations due to their "generate-and-test" approach, which is highlighted in [Trinh et al. 2014].

Unbounded methods are often built upon the theory of automata or regular languages. We will be brief and mention a few notable works. Java String Analyzer [Christensen et al. 2003] applies static analysis to model flow graphs of Java programs in order to capture constraints among string variables. A finite automata is then derived to constrain possible string values. The work [Shannon et al. 2009] used finite state machines (FSMs) for abstracting strings during symbolic execution of Java programs. They handle a few core methods in the `java.lang.String` class and some other related classes. They partially integrate a numeric constraint solver. For instance, string operations which return integers, such as `indexOf`, trigger case-splits over all possible return values. Recent automata-based string solvers such as Norn [Abdulla et al. 2015], ABC [Aydin et al.

2015], Trau [Abdulla et al. 2017, 2019], Sloth [Holik et al. 2017], Ostrich [Chen et al. 2019], have made a great effort towards efficient handling of length constraints, yet their practical application is still limited.

Most of recent works on string solving are based on unbounded methods with string as a primitive data type. Examples are CVC4 [Liang et al. 2014; Reynolds et al. 2019, 2017], S3 [Trinh et al. 2014, 2016, 2017], Z3-str [Zheng et al. 2013], Z3-str2 [Zheng et al. 2015], Z3-str3 [Berzish et al. 2017]. These solvers have shown the applicability and efficiency in symbolic execution of web programs. We will briefly discuss most recent works such as Z3-str3 [Berzish et al. 2017] and CVC4 [Reynolds et al. 2019, 2017]. Z3str3 introduces theory-aware branching heuristics, where they modify Z3's branching heuristic to take into account the structure of theory literals to compute branching activities. However, the heuristics only affect the decision on which branch to proceed. Instead, our analysis focuses more on the construction of an efficient search tree by performing "splitting" *smartly*. This is also guided by the inter-theory dependency between string equations and length (and extended string) constraints. CVC4 proposes context-dependent simplification rules to reduce extended string constraints. Although this technique shows the efficiency in practice, such set of rules is incomplete. On the other hand, this paper contributes directly to the handling of the core fragment. Our work, therefore, can naturally complement CVC4 techniques; studying this is left as future work.

8 CONCLUSION

We have presented our dependency analysis to deal with the interaction among constraints from different theories. The analysis not only takes into account the complexity of inter-theory dependency but also the complexity of resolving such dependency. This leads to the construction of a partial order for string equations where length constraints and extended string constraints are used to drive the construction. In the SMT setting, this enables us to embed a better search strategy into the core case-splitting step. We have implemented the analysis in the native string theory solver of Z3, evaluated our new solver experimentally and shown that it can decide the satisfiability for more benchmarks, and in doing so, exhibits substantial speedup against other state-of-the-art solvers. We believe the techniques are also applicable to other domains such as sequence, set/multi-set, where there is also interaction between multiple theories.

ACKNOWLEDGMENTS

We warmly thank the anonymous OOPSLA reviewers for helping us improve the presentation. This research is partly supported by the National Research Foundation, Prime Minister's Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme.

REFERENCES

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukas Holik, Ahmed Rezine, and Philipp Rummer. 2017. Flatten and Conquer: A Framework for Efficient Analysis of String Constraints. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 602–617. https://doi.org/10.1145/3062341.3062384

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. 2015. Norn: An SMT Solver for String Constraints. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 462–469.

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bui Phi Diep, Lukáš Holík, and Petr Janků. 2019. Chain-Free String Constraints. In *Automated Technology for Verification and Analysis*, Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza (Eds.). Springer International Publishing, Cham, 277–293.

Roland Axelsson, Keijo Heljanko, and Martin Lange. 2008. Analyzing Context-Free Grammars Using an Incremental SAT Solver. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming, Part II (ICALP '08)*. Springer-Verlag, Berlin, Heidelberg, 410–422. https://doi.org/10.1007/978-3-540-70583-3_34

Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. 2015. Automata-Based Model Counting for String Constraints. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 255–272.

Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. 2017. Z3str3: A String Solver with Theory-Aware Heuristics. In *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design (FMCAD '17)*. FMCAD Inc, Austin, Texas, 55–59.

Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. 2009. Path Feasibility Analysis for String-Manipulating Programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, Stefan Kowalewski and Anna Philippou (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 307–321.

J. Richard Büchi and Steven Senger. 1990. *Definability in the Existential Theory of Concatenation and Undecidable Extensions of this Theory*. Springer New York, New York, NY, 671–683. https://doi.org/10.1007/978-1-4613-8928-6_37

Taolue Chen, Matthew Hague, Anthony W. Lin, Philipp Rummer, and Zhilin Wu. 2019. Decision Procedures for Path Feasibility of String-Manipulating Programs with Complex Operations. *Proc. ACM Program. Lang.* 3, POPL, Article 49 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290362

Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. 2003. Precise Analysis of String Expressions. In *Proceedings of the 10th International Conference on Static Analysis (SAS'03)*. Springer-Verlag, Berlin, Heidelberg, 1–18.

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

Vijay Ganesh, Adam Kieżun, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael Ernst. 2011. HAMPI: A String Solver for Testing, Analysis and Vulnerability Detection. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–19.

Vijay Ganesh, Mia Minnes, Armando Solar-Lezama, and Martin Rinard. 2013. Word Equations with Length Constraints: What's Decidable?. In *Hardware and Software: Verification and Testing*, Armin Biere, Amir Nahir, and Tanja Vos (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 209–226.

Jun He, Pierre Flener, Justin Pearson, and Wei Ming Zhang. 2013. Solving String Constraints: The Case for Constraint Programming. In *Principles and Practice of Constraint Programming*, Christian Schulte (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 381–397.

Lukas Holik, Petr Janku, Anthony W. Lin, Philipp Rummer, and Tomas Vojnar. 2017. String Constraints with Concatenation and Transducers Solved Efficiently. *Proc. ACM Program. Lang.* 2, POPL, Article 4 (Dec. 2017), 32 pages. https://doi.org/10.1145/3158092

Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. 2014. A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 646–662.

Anthony W. Lin and Rupak Majumdar. 2018. Quadratic Word Equations with Length Constraints, Counter Systems, and Presburger Arithmetic with Divisibility. In *Automated Technology for Verification and Analysis*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 352–369.

G. S. Makanin. 1977. THE PROBLEM OF SOLVABILITY OF EQUATIONS IN A FREE SEMIGROUP. *Mathematics of the USSR-Sbornik* 32, 2 (1977), 129.

OWASP. 2013. Top ten project. http://www.owasp.org/.

Gideon Redelinghuys, Willem Visser, and Jaco Geldenhuys. 2012. Symbolic Execution of Programs with Strings. In *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference (SAICSIT '12)*. Association for Computing Machinery, New York, NY, USA, 139–148. https://doi.org/10.1145/2389836.2389853

Andrew Reynolds, Andres Nötzli, Clark Barrett, and Cesare Tinelli. 2019. High-Level Abstractions for Simplifying Extended String Constraints in SMT. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 23–42.

Andrew Reynolds, Maverick Woo, Clark Barrett, David Brumley, Tianyi Liang, and Cesare Tinelli. 2017. Scaling Up DPLL(T) String Solvers Using Context-Dependent Simplification. In *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 453–474. https://doi.org/10.1007/978-3-319-63390-9_24

Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A Symbolic Execution Framework for JavaScript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP '10)*. IEEE Computer Society, USA, 513–528. https://doi.org/10.1109/SP.2010.38

D. Shannon, I. Ghosh, S. Rajan, and S. Khurshid. 2009. Efficient Symbolic Execution of Strings for Validating Web Applications. In *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009) (DEFECTS '09)*. Association for

Computing Machinery, New York, NY, USA, 22–26. https://doi.org/10.1145/1555860.1555868

Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2014. S3: A Symbolic String Solver for Vulnerability Detection in Web Applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. Association for Computing Machinery, New York, NY, USA, 1232–1243. https://doi.org/10.1145/2660267.2660372

Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2016. Progressive Reasoning over Recursively-Defined Strings. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 218–240.

Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2017. Model Counting for Recursively-Defined Strings. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 399–418.

Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Julian Dolby, and Xiangyu Zhang. 2015. Effective Search-Space Pruning for Solvers of String Equations, Regular Expressions and Length Constraints. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 235–254.

Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-Str: A Z3-Based String Solver for Web Application Analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 114–124. https://doi.org/10.1145/2491411.2491456