Form Methods Syst Des (2017) 50:249–288
DOI 10.1007/s10703-016-0263-6

Z3str2: an efficient solver for strings, regular expressions, and length constraints

Yunhui Zheng[1] · Vijay Ganesh[3] · Sanu Subramanian[3] · Omer Tripp[2] · Murphy Berzish[3] · Julian Dolby[1] · Xiangyu Zhang[4]

Published online: 27 December 2016

Abstract In recent years, string solvers have become an essential component in many formal verification, security analysis, and bug-finding tools. Such solvers typically support a theory of string equations, the length function, and the regular-expression membership predicate. These enable considerable expressive power, which comes at the cost of slow solving time, and in some cases even non-termination. We present three techniques, designed for word-based SMT string solvers, to mitigate these problems: (1) detecting overlapping variables, which is essential to avoiding common cases of non-termination; (2) pruning of the search space via bi-directional integration between the string and integer theories, enabling new

O. Tripp: The research leading to this manuscript, and earlier versions of the manuscript, were completed while the author was at IBM Research.

✉ Yunhui Zheng
zhengyu@us.ibm.com

Vijay Ganesh
vganesh@uwaterloo.ca

Sanu Subramanian
ses@uwaterloo.ca

Omer Tripp
trippo@google.com

Murphy Berzish
mtrberzi@uwaterloo.ca

Julian Dolby
dolby@us.ibm.com

Xiangyu Zhang
xyzhang@cs.purdue.edu

1 IBM T. J. Watson Research Center, Yorktown Heights, NY, USA

2 Google, Mountain View, CA, USA

3 University of Waterloo, Waterloo, ON, Canada

4 Purdue University, West Lafayette, IN, USA

cross-domain heuristics; and (3) a binary search based heuristic, allowing the procedure to skip unnecessary string length queries and converge on consistent length assignments faster for large strings. We have implemented above techniques atop the Z3-str solver, resulting in a significantly more robust and efficient solver, dubbed Z3str2, for the quantifier-free theory of string equations, the regular-expression membership predicate, and linear arithmetic over the length function. We report on a series of experiments over four sets of challenging real-world benchmarks, where we compare Z3str2 with five different string solvers: S3, CVC4, Kaluza, PISA and Stranger. Each of these tools utilizes a different solving strategy and/or string representation (based e.g. on words, bit vectors or automata). The results point to the efficacy of our proposed techniques, which yield dramatic performance improvement. We also demonstrate performance improvements enabled by Z3str2 in the context of symbolic execution for string-manipulating programs. We observe that the techniques presented here are of broad applicability, and can be integrated into other string solvers to improve their performance.

Keywords String constraint solver · SMT solver · String analysis

1 Introduction

Reasoning over strings is becoming increasingly important due to security threats imposed by improper handling of untrusted string values [8,23,32]. In response, different powerful string solvers have been developed, including HAMPI [23], Kaluza [33], PISA [35], Stranger [38], CVC4 [26], S3 [36], Norn [6] and Z3-str [41]. These tools primarily solve the satisfiability problem over string (aka word) equations, with some of them also providing support for regular expression (RE) membership predicates and linear arithmetic over the length function (e.g. [26,36]). While these tools have improved dramatically in recent years, the demand for even more efficient solvers continues to grow unabated as software continues to grow in size and complexity.

Motivated by this need for efficient string solvers, we present two new techniques to solve combined string, regular expression, and integer constraints. These techniques are applicable primarily to SMT solvers that treat strings without abstractions or representation conversions, which we refer to collectively as *word-based string solvers*. Examples of such solvers include the Z3-str, CVC4 and S3 string solvers.

For the sake of completeness, we compare and contrast our techniques against solvers that use automata (e.g., PISA and Stranger) and bit-vector (e.g., Kaluza) string representations. Word-based string solvers have several important advantages: First, unlike bit-vector-based solvers, they can precisely model unbounded strings and string equalities without over-approximation, a feature that is crucial to string analysis of web applications. Second, by modeling strings and length in native domains, word-based string solvers can leverage the state of the art in integer constraint solving, and further enable hybrid techniques via powerful SMT engines. Finally, such solvers can take advantage of well-developed application-specific rewrite rules.

At the same time, a fundamental problem of word-based string solvers (unlike those based on bit vectors, which impose a finite domain) is that it is unclear, at present, whether the satisfiability problem for the quantifier-free theory of word equations, regular-expression membership predicate, and length function is decidable [28]. All current practical string solvers suffer from incompleteness and nontermination. Addressing these problems is of primary importance, calling for new techniques to effectively explore the solution space. In light of this motivation, we have developed three techniques that address the problems of nontermination and search-space explosion.

First, a well-known reason for nontermination is overlapping variables [6,14,41], as we illustrate with the equation $a \cdot X = X \cdot b$, where a, b are constant strings and X is a string variable. Stated intuitively, the solution for X has to be in the form of $a \cdot X_1 \cdot b$, where X_1 is a string variable. The reduction step results in $a \cdot X_1 = X_1 \cdot b$, which is in the same form as the original equation, and thus leads to nontermination. However, this equation is obviously unsatisfiable. We revisit this example in Sect. 4, which highlights the need for a robust procedure to detect overlapping variables.

The second technique, given the tight interplay between string and integer values (in index-sensitive string operations), is bi-directional solver-level integration between the string and integer theories. This can be leveraged to drastically reduce the search space for typical constraints obtained from practical applications.

The last technique is a binary search based heuristic that helps to achieve faster convergences in selecting consistent length assignment for large strings. As we observe, a huge number of length assignment polling iterations are redundant. We adapt a binary search algorithm and search the unbounded length space more efficiently.

We have implemented all of these techniques atop the Z3-str solver as the Z3str2 solver for the satisfiability problem over a quantifier-free theory of word equations, regular-expression membership predicate as well as the length function. We report on a comprehensive set of experiments that validate the efficacy of our proposed techniques by comparing Z3str2 with Kaluza, PISA, Stranger, S3 and CVC4 over four sets of benchmarks derived from the real world.[1] Besides, we illustrate performance improvements in the context of symbolic execution of string manipulating programs. We show that we are able to avoid exploring a large number of execution paths while precisely capturing the semantics of common string utility functions. We emphasize that our techniques are applicable also to other word-based string solvers such as S3 and CVC4.

Contributions To summarize, this paper makes the following principal contributions:

1. *Formal representation of solving word equations* We use a boundary label based representation to formally model the key steps of solving word equations.
2. *Guided search* We present three techniques designed for string solvers that treat strings as primitive types. The first is a sound and complete method to detect overlapping variables, which improves performance and avoids exploration of certain paths that may lead to nontermination. The second technique is a two-way integration between the string and integer theories, which enables effective pruning based on cross-domain heuristics. The last one is a binary search based pruning heuristic that allows the procedure to skip redundant string length assignment queries.
3. *Z3-str integration* We have integrated all of the aforementioned techniques into the core solving algorithm of Z3-str. We describe the architecture of the resulting tool, Z3str2, and prove its soundness.
4. *Symbolic execution performance improvements* We demonstrate that we can significantly simplify the constraint encoding/solving procedure and improve the performance of symbolic execution on string manipulating programs.
5. *Experimental study* To validate the efficacy of our techniques, we have conducted a comprehensive set of experiments where we compare Z3str2 against five solvers—namely, S3, CVC4, Kaluza, PISA and Stranger—on four benchmark suites. The results show that

[1] The Z3str2 code, as well as the data pertaining to our experiments, are all available at [1].

Z3str2 is significantly faster than competing solvers (often by orders of magnitude) in all but few cases.

2 Preliminaries

In this section, we briefly introduce the syntax and semantics of the quantifier-free theory of word equations, regular expression membership, and length over integers. This theory will be the basis of other formalizations presented in this paper.
Syntax of word equations, RE membership, and length We fix a disjoint two-sorted set of variables $var = var_{str} \cup var_{int}$; var_{str} consists of string variables, denoted $X, Y, S, \ldots$ and var_{int} consists of integer variables, denoted $m, n, \ldots$. We also fix a two-sorted set of constants $Con = Con_{str} \cup Con_{int}$. Moreover, $Con_{str} \subset \Sigma^*$ for some finite alphabet, Σ, whose elements are denoted $f, g, \ldots$. Elements of Con_{str} will be referred to as *string constants* or *strings*. Elements of Con_{int} are nonnegative integers.[2] The empty string is represented by ϵ, and has length 0. Terms may be string terms or integer terms. A string term is either an element of var_{str}, an element of Con_{str}, or a concatenation of string terms (denoted by the function *concat* or interchangeably by $\cdot$). An integer term is an element of var_{int}, an element of Con_{int}, the length function applied to a string term, or an addition, subtraction, multiplication, or division of two integer terms. The theory contains three types of atomic formulas, namely, word equations, length constraints, and regular expression (RE) membership predicates. REs are defined inductively, where constants and the empty string form the base case, and the operations of concatenation, alternation, and Kleene star are used to build up more complicated expressions (see details in [17]). REs may not contain variables. Z3str2 supports a list of common string-related operators such as charAt, contains, startswith, endswith, indexof, lastindexof, substring and etc. They are desugared to word equations with length functions. Formulas are defined inductively over atomic formulas and are quantifier-free. Figure 1 summarizes the syntax of the constraint language.
Semantics of word equations, RE membership, and length For a word w, $len(w)$ denotes the length of w. The universe of discourse for the str sort is the set of strings Σ^*, and for the int sort it is the set of natural numbers. For a word equation $t_1 = t_2$, we refer to t_1 as the left hand side (LHS), and t_2 as the right hand side (RHS). We fix a finite alphabet Σ of characters over which strings are defined. Given a formula θ, an *assignment* for θ (with respect to Σ) is a map from the set of variables in θ to $\Sigma^* \cup \mathbb{N}$ (where string variables are mapped to strings and integer variables are mapped to numbers). Given such an assignment, θ can be interpreted as an assertion about Σ^* and $\mathbb{N}$. If this assertion is true, then θ is *satisfiable* or SAT. A formula with no satisfying assignment is *unsatisfiable* or UNSAT. Two formulas θ, ϕ are *equisatisfiable* if the assertion "θ is SAT iff ϕ is SAT" holds. The *satisfiability problem* for a set S of formulas is to decide whether any given formula in S is SAT or not.
Soundness, completeness, termination, decidability

A satisfiability procedure is *sound* if, whenever the procedure answers UNSAT, then the input is indeed unsatisfiable. Conversely, a satisfiability procedure is *complete* if, whenever the input is satisfiable, the procedure will answer SAT.

[2] This restriction is made only for consistency with the SMT-LIB standard [7] in a typical implementation. The only integer literals allowed in the SMT-LIB syntax are 0 and positive integers. Note that both the SMT-LIB standard and the syntax we give here still allow negative integer constants to be expressed as, for example, `(- 0 3)` for -3.

```
Term:bool     ::=    Var:bool
                  |  true
                  |  false
                  |  (Contains  Term:string  Term:string)
                  |  (StartsWith  Term:string  Term:string)
                  |  (EndsWith  Term:string  Term:string)
                  |  (RegexIn  Term:string  Term:regex)
Term:int      ::=    Var:int
                  |  Number
                  |  ({+, −, ×, ÷}  Term:int  Term:int)
                  |  (Length  Term:string)
                  |  (IndexOf  Term:string  Term:string)
                  |  (IndexOf2  Term:string  Term:string  Term:int)
                  |  (LastIndexOf  Term:string  Term:string)
Term:string   ::=    Var:string
                  |  ConstStr
                  |  (Concat  Term:string  Term:string)
                  |  (Substring  Term:string  Term:int  Term:int)
                  |  (Replace  Term:string  Term:string  Term:string)
                  |  (CharAt  Term:string  Term:int)
Term:regex    ::=    (Str2Regex  ConstStr:string)
                  |  (RegexStar Term:regex )
                  |  (Term:regex)+
                  |  (Term:regex)?
                  |  (RegexConcat  Term:regex  Term:regex)
                  |  (RegexUnion  Term:regex  Term:regex)
Expr:bool     ::=    Term:bool
                  |  (=  Term:bool  Term:bool)
                  |  (not  Expr:bool)
                  |  (and  Expr:bool  Expr:bool)
                  |  (or  Expr:bool  Expr:bool)
                  |  (ite  Expr:bool  Expr:bool  Expr:bool)
                  |  (implies  Expr:bool  Expr:bool)
                  |  ({<, ≤, =, ≥, >}  Term:int  Term:int)
                  |  (=  Term:string  Term:string)
Assertion     ::=    (assert  Expr:bool )
```

Fig. 1 Constraint syntax

An algorithm is *terminating* if it returns an answer in finite time on every input.

A satisfiability procedure is a *decision procedure* if and only if it is sound, complete, and terminating. The satisfiability problem for a set of formulas is decidable if and only if there exists a decision procedure that solves its satisfiability problem.

3 Overview of the Z3str2 string solver

The Z3str2 solver is a string plug-in built into the Z3 SMT Solver [12], with an efficient integration between the string plug-in and Z3's integer solver. As can be seen from the architectural schematic of the Z3str2 string solver given in Fig. 2 (the algorithm is explained in detail in Sect. 6.2.3), the first step is to purify the input into two parts, namely string constraints (word equations and RE membership) on the one hand, and integer linear arithmetic constraints over the length function on the other hand. Next, the word equations and RE constraints are given as input to the string plug-in. The plug-in may consult the Z3 core to detect equivalent terms. The word equations are solved using an algorithm described in detail in Sect. 4 below. The RE constraints are solved by unrolling as described also in Sect. 4. The length constraints are converted into a system of pure integer linear arithmetic inequations

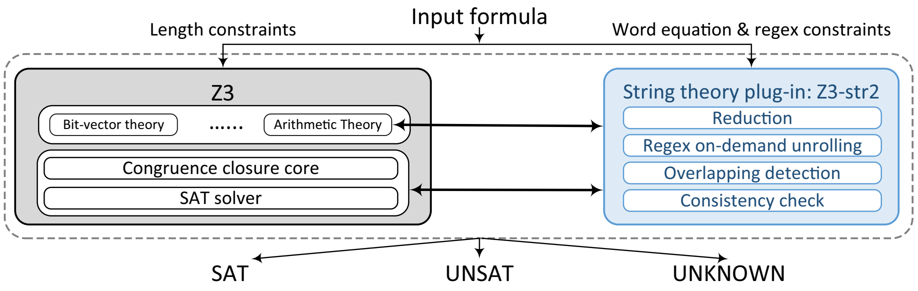


Fig. 2 Architecture of Z3str2

and solved using Z3's integer solver. During the solving process, the string plug-in may generate length constraints that are incrementally added on demand to Z3's integer solver, which are regularly checked for consistency with both the input length constraints and any previously generated ones.

On any well-formed input, as described in Sect. 2, Z3str2 may return SAT, UNSAT or UNKNOWN. Note that while Z3str2 can handle a boolean combination of atomic formulas, we refer only to conjunctions of literals in the rest of paper without loss of generality. Z3str2, like other competing solvers such as CVC4, is sound but not complete. If either Z3's integer solver or our string plug-in determines that their respective purified inputs are UNSAT, Z3str2 reports UNSAT. If the string plug-in detects that the input equations have complicated overlaps that its heuristics cannot handle, it reports UNKNOWN. This is a source of incompleteness in Z3str2's implementation.

The third and only remaining possibility is that both Z3's integer solver and the string plug-in determine that their respective purified inputs are SAT. However, this does not necessarily mean that the input is SAT. It could be that the solution produced by the integer solver is inconsistent with the solution produced by the string solver, or vice versa. For example, the integer solver may say that a particular string variable, say X, has length equal to 1, while the string plug-in might produce a specific assignment for X that is of length equal to 2. One way to overcome this problem is to iterate through all possible solutions until a consistent one is found, assuming one exists. However, given that the domain of strings and natural numbers is infinite, it is possible that such an iterative procedure may loop forever in the event there are no consistent solutions to be found. In other words, if the input is indeed SAT, the procedure discussed here will correctly establish consistency and determine that the input is SAT. Unfortunately, it is possible that, when the input is in fact UNSAT, both the integer solver and string plug-in may determine that their respective purified inputs are SAT and may then loop forever searching for a combined consistent solution. They are obviously not going to find a combined consistent solution in such cases given that the input is in fact UNSAT, and hence the iterative procedure may not terminate.

The above-described problem of non-termination due to the interaction between the integer and string parts of the theory is not specific to Z3str2. In fact, the problem of deciding the satisfiability problem for the quantifier-free theory of word equations and length function remains open after decades of research and is a major open problem in mathematical logic [28]. In conclusion, if Z3str2 reports that the input is UNSAT, then indeed the input is UNSAT (soundness). However, the converse is not necessarily true, i.e., just like all other competing practical solvers, Z3str2 is not complete.

4 Word equation sub-solver in Z3str2

In this subsection, we focus on the word equation solving component of Z3str2. Starting with the work of Makanin [27], many decision procedures for word equations have been proposed [19,30,34]. While most procedures are not accompanied by practical implementations, they are a rich source of ideas for all the solvers that have recently been implemented. For example, the Z3str2 solver is based on the concepts of *boundary labels, generalized word equations and arrangements* (discussed in greater detail in Sect. 6), that have their roots in the very first decision procedure for word equations by Makanin.

The key technique used by Z3str2 to solve a word equation W is to recursively convert W equisatisfiably into a disjunction of conjunctions of simpler equations, which we call "arrangements". These arrangements are computed by aligning the concatenation function on the LHS and RHS of a given equation such that an occurrence of concatenation function in the LHS (resp. RHS) may "split" or "cut" variables on the RHS (resp. LHS).

As an illustration, consider the following formula composed of three equations: $Z = X \cdot Y \;\wedge\; Z = W \cdot c \;\wedge\; c \cdot Y = c \cdot b \cdot c$, where X, Y, Z and W are string variables, and b and c are characters. A simple rewriting is the following:

$$\begin{aligned} Z = X \cdot Y &\Longrightarrow\; Z_1 = X \;\wedge\; Z_2 = Y^{\,[1.1]} \\ Z = W \cdot c &\Longrightarrow\; Z_1 = W \;\wedge\; Z_2 = c^{\,[1.2]} \\ c \cdot Y = c \cdot b \cdot c &\Longrightarrow\; Y = b \cdot c^{\,[1.3]} \end{aligned}$$

Observe that Z is split into Z_1 and Z_2, which are constrained differently. However, this rewriting is not satisfiable because $Y = c$ from equations [1.1] and [1.2], and $Y = b \cdot c$ from equation [1.3]. Now observe that the alignment described above is not the only one we can consider. Below is a different alignment that leads to a new splitting and in fact yields a satisfying assignment:

$$Z = W \cdot c \Longrightarrow\; Z_1 = W_1^{[1.4]} \;\wedge\; Z_2 = W_2 \cdot c^{\,[1.5]}$$

The difference now is that W is also split (into W_1 and W_2), and hence this splitting yields a satisfying assignment. In particular, from [1.1], [1.3] and [1.5], we have $W_2 = b$. Also note that X, Z_1 and W_1 become free variables as they are all equivalent but not constrained by any other variable.

What the above example highlights is that there are many different alignments of variable boundaries in the LHS (resp. RHS) that can split variables in the RHS (resp. LHS). We call every such alignment an *arrangement*. Here is the crucial fact about word equations: every equation can be equisatisfiably rewritten into a finite set of arrangements, where each arrangement is a finite set of word equations obtained from the splitting procedure described above. The Z3str2 solver exploits this fact, and solves word equations by converting them into finite sets of arrangements and inspecting each one individually to see if they are satisfiable. The input word equation is SAT if and only if at least one arrangement is SAT. This in a nutshell is how the word equations are solved by the Z3str2 solver, i.e., by recursively converting equations into a disjunction of arrangements (where each arrangement is a simpler set of equations) until a set of arrangements is derived where the satisfiability can determined purely via inspection.

5 Supporting regular expression membership predicates

A RE membership predicate $X \in \mathcal{R}$ is reduced to word equations by a transformation function $\rho(X, \mathcal{R})$, where X is a string variable and $\mathcal{R}$ is a regular expression. The function is defined as follows:

$$\begin{array}{ll}
\rho(X, s) & ::= X = s, \mathit{where\ s\ is\ a\ constant\ string} \\
\rho(X, \mathcal{R}_1 | \mathcal{R}_2) & ::= \rho(X, \mathcal{R}_1) \vee \rho(X, \mathcal{R}_2) \\
\rho(X, \mathcal{R}_1 \cdot \mathcal{R}_2) & ::= X = T_1 \cdot T_2 \wedge \rho(T_1, \mathcal{R}_1) \wedge \rho(T_2, \mathcal{R}_2) \\
\rho(X, \mathcal{R}^*) & ::= X = \mathit{unroll}(\mathcal{R}, n) \wedge n \geq 0
\end{array}$$

where n is a fresh integer variable for each Kleene star operation; T_1 and T_2 are fresh string variables; $\mathit{unroll}(\mathcal{R}, n)$ represents the expression obtained by unrolling $\mathcal{R}$ n times. After the RE membership predicates are replaced by word equations, the string solver proceeds as usual. When the solver explores various arrangements, the $\mathit{unroll}()$ functions are further simplified by the following rules.

$$\begin{array}{rl}
X = \mathit{unroll}(\mathcal{R}, n_1) & ::= \textbf{if } (n_1 = 0) \textbf{ then } \{X = \epsilon\} \textbf{ else } \{X = T_3 \cdot \mathit{unroll}(\mathcal{R}, n_1 - 1) \wedge \rho(T_3, \mathcal{R})\} \\
X \cdot Y = \mathit{unroll}(\mathcal{R}, n_1) & ::= \textbf{if } (n_1 = 0) \textbf{ then } \{X = \epsilon \wedge Y = \epsilon\} \textbf{ else } \{X = \mathit{unroll}(\mathcal{R}, n_2) \cdot T_3 \\
& \quad \wedge Y = T_4 \cdot \mathit{unroll}(\mathcal{R}, n_3) \wedge n_1 = n_2 + n_3 + 1 \wedge \rho(T_3 \cdot T_4, \mathcal{R})\}
\end{array}$$

Note that $\mathcal{R}$ is essentially unrolled once in the else branch of both rules. Just like in other existing solvers that support RE, the unrolling process may not terminate, especially when there are no length constraints associated with the involved variables. We hence rely on a timeout mechanism.

6 Solving word equations and overlapping variable detection

In this section, we formally explain the key steps of solving word equations. While simple and elegant for typical equations obtained from program analysis, the word equation solver described here may fall into infinite loops under certain circumstances.

We first use two examples to show the challenges introduced by overlapping variables. In fact, the problem of "overlapping" variables described below is recognized by logicians as the crucial source of complexity in solving word equations. Then, we describe how we solve word equations and avoid the nontermination brought by overlapping variables.

6.1 Examples that highlight the crucial overlap detection problem

Example 1 Formula reduction may fall into infinite loops. Consider the following formula which was introduced in Sect. 1.

$$a \cdot X = X \cdot b$$

where X is a string variable.

In this example, X appears both as the LHS suffix and as the RHS prefix. This equation is not satisfiable. However, if we solve it using the word equation procedure described in Sect. 4, then the algorithm will not terminate. In particular, the equation has three arrangements. The first arrangement is where $X = \epsilon$, resulting in the equation $a = b$ which is unsatisfiable. The second arrangement is where the concatenation function in the LHS and RHS align exactly such that we get $X = a \wedge X = b$. The third arrangement is where the LHS occurrence of X cuts the RHS occurrence in the RHS illustrated in Fig. 3.

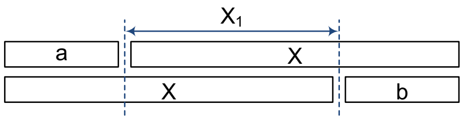


Fig. 3 Graphical representation of the input formula. The *top row* is the LHS and the *bottom row* is the RHS. *Each box* represents a substring. The diagram means that the two occurrences of X overlap

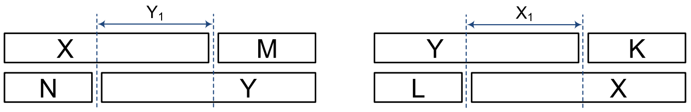


Fig. 4 Graphical representation of one arrangement of the input formula in Example 2. Both X and Y occur twice and they overlap in different ways

Note that the suffix of X that occurs in the RHS (bottom part of Fig. 3) of the equation overlaps with the prefix of X in the LHS (top part of Fig. 3). We represent this overlapping part with a new variable X_1. By applying some simple rewrites we derive the following:

$$a \cdot X = X \cdot b \implies X = a \cdot X_1^{[2.1]} \wedge X = X_1 \cdot b^{\,[2.2]}$$

From [2.1] and [2.2], we can infer $a \cdot X_1 = X_1 \cdot b$. Note this derived equation has the same form as the input formula. As a result, the above-mentioned procedure will not terminate, unless some steps are taken to detect such “overlaps” and determine their satisfiability without computing arrangements ad infinitum. □

One could imagine heuristics to detect and handle relatively simple overlaps described above. However, in general, when many equations are involved with variables overlapping indirectly, the problem is not easy to detect or decide.

Example 2 The previous example shows a simple non-terminating case. In practice, non-terminating cases can be very complex, involving multiple equations. Consider the following formula.

$$X \cdot M = N \cdot Y \quad \wedge \quad Y \cdot K = L \cdot X$$

where X, Y, M, N, K, and L are all string variables.

The following rewriting shows why a naive solving procedure will not terminate for such an input.

Similarly to the arrangements presented in Example 1, Fig. 4 shows one possibility of both equations. The following formulas are the derived equations corresponding to the graphical representations in Fig. 4.

$$X \cdot M = N \cdot Y \implies X = N \cdot Y_1^{[3.1]} \wedge Y = Y_1 \cdot M^{\,[3.2]}$$
$$Y \cdot K = L \cdot X \implies Y = L \cdot X_1^{[3.3]} \wedge X = X_1 \cdot K^{\,[3.4]}$$

The smaller equations can be rewritten as follows.

$$[3.1] \wedge [3.4] \implies N \cdot Y_1 = X_1 \cdot K^{\,[3.5]}$$
$$[3.2] \wedge [3.3] \implies L \cdot X_1 = Y_1 \cdot M^{\,[3.6]}$$

Formula [3.7] and [3.8] can be derived from an arrangement of [3.5]. Similarly, [3.9] and [3.10] represent one possible arrangement of [3.6].

$$[3.5] \implies X_1 = N \cdot Y_2^{[3.7]} \wedge Y_1 = Y_2 \cdot K^{\,[3.8]}$$

Fig. 5 Definitions

$$\begin{array}{lcl}
Formula\ \mathbb{F} & := & \mathbb{E}_1 \wedge \mathbb{E}_2 \wedge ...\mathbb{E}_n \\
WordEquation\ \mathbb{E} & := & \mathcal{W}_1 = \mathcal{W}_2 \\
Character\ c & := & \{a, b, ...\} \\
String\,Variable\ X & := & \{X, Y, ...\} \\
Word\ \mathcal{W} & := & {}^{L}\varepsilon \mid {}^{L}c\mathcal{W} \mid {}^{L}X\mathcal{W} \\
Label\ l & := & \{\rhd_1^X, \lhd_1^X, \rhd_2^c, \lhd_2^c, ...\} \\
LabelSet\ L & := & \mathcal{P}(Label)
\end{array}$$

$$[3.6] \Longrightarrow Y_1 = L \cdot X_2{}^{[3.9]} \wedge X_1 = X_2 \cdot M^{\ [3.10]}$$

The following new equivalences are derived.

$$[3.7] \wedge [3.10] \Longrightarrow X_2 \cdot M = N \cdot Y_2{}^{[3.11]}$$
$$[3.8] \wedge [3.9] \Longrightarrow Y_2 \cdot K = L \cdot X_2{}^{[3.12]}$$

Observe that formula ([3.11] ∧ [3.12]) is in the exact same form as the input. Since the loop appears after more than one round of reduction, we use the term "*multiple layer overlaps*" to denote such cases. A key contribution of our procedure is to detect and handle such overlaps. □

In fact, overlapping variables get to the crux of the difficulty of solving word equations, for otherwise simple rewrites can solve such equations. Hence, any solution to detecting overlaps and deciding such equations is of universal value, and can be used as a subroutine by different types of string solvers. In the following subsections, we discuss a solution to the overlap detection problem.

6.2 Formal description

Here we formally describe the procedure for solving word equations.

Figure 5 recursively defines the formulas in the quantifier-free theory of word equations that are input to the procedure we describe later. These definitions describe the syntax of equations. A formula $\mathbb{F}$ is a conjunction of a set of word equations $\mathbb{E}$.[3] An equation in $\mathbb{E}$ dictates the equivalence of two words. A word $\mathcal{W}$ is a sequence of string variables (e.g. X) or constant characters (e.g. c) delimited by label sets. Labels uniquely identify the boundaries of an occurrence of a character or a variable. Note that a character (variable) may occur multiple times in a formula. We use $\rhd_1^X$ and $\lhd_1^X$ to describe the left and right labels of the first occurrence of X respectively, following a pre-defined initial order. Below we show a sample equation.

$$^{\{\rhd_1^a\}}a^{\{\lhd_1^a, \rhd_1^X\}}X^{\{\lhd_1^X\}} = {}^{\{\rhd_2^X\}}X^{\{\lhd_2^X, \rhd_1^b\}}b^{\{\lhd_1^b\}} \qquad [\mathbb{E}_1]$$

Note that labels are introduced to facilitate formal definition of the procedure. The input equations provided by users do not contain any labels. Henceforth, we represent concatenation explicitly through the '·' operator or implicitly as in the definition of *Word*, if doing so does not cause any misunderstanding.

[3] While we do support AND–OR combination of word equations, without loss of generality it is sufficient to describe the procedure only for a conjunction of word equations.

6.2.1 Label arrangements

Boundary labels are particularly important for our technique. Intuitively, we leverage them to reason about the relative positions of the sub-parts in words, such that we can reduce the original equations to a set of smaller equations for the corresponding sub-parts until the equations become so fine-grained that the solution can be directly inferred. The input system of equations is UNSAT if none of the possible break-downs leads to valid solutions. In this subsection, we explain how our technique manipulates labels. We discuss how to split equations to smaller ones based on the arrangements and how to determine if equations are in *solvable form* in the next subsection.

We now formalize the concept of *label arrangement* to facilitate our discussion. A label arrangement, or arrangement in short, is a sequence of label sets as defined in Fig. 6. Each label set may contain labels from multiple words, such as the LHS and RHS words of an equation. Two labels in the same set imply the corresponding boundaries *align*. For example, assume the following equation $[\mathbb{E}_2]$.

$$^{\{\triangleright_1^a\}}a^{\{\triangleleft_1^a, \triangleright_1^Y\}}Y^{\{\triangleleft_1^Y\}} = {}^{\{\triangleright_1^X\}}X^{\{\triangleleft_1^X, \triangleright_1^b\}}b^{\{\triangleleft_1^b\}} \qquad [\mathbb{E}_2]$$

One arrangement of its LHS and RHS words is the following.

$$\{\triangleright_1^a, \triangleright_1^X\} \cdot \{\triangleleft_1^a, \triangleleft_1^X, \triangleright_1^Y, \triangleright_1^b\} \cdot \{\triangleleft_1^Y, \triangleleft_1^b\} \qquad [\mathcal{A}_1]$$

The corresponding alignment (with the labels) can be graphically illustrated in Fig. 7a.

$\mathit{LabelArrangement}\ \mathcal{A} \ :=\ \overline{\mathcal{P}(\mathit{Label})}$

ArrgmtProduct $\otimes$: $\mathit{LabelArrangement} \times \mathit{LabelArrangement} \longrightarrow \mathcal{P}(\mathit{LabelArrangement})$

$$\begin{aligned}
L_1 \cdot L_2 \cdot \ldots \cdot L_n \otimes R_1 \cdot R_2 \cdot \ldots \cdot R_m &:= L_1 \cup R_1 \cdot (L_2 \cdot \ldots \cdot L_{n-1} \,\widetilde{\otimes}\, R_2 \cdot \ldots \cdot R_{m-1}) \cdot L_n \cup R_m \\
L_1 \cdot L_2 \cdot \ldots \cdot L_n \,\widetilde{\otimes}\, R_1 \cdot R_2 \cdot \ldots \cdot R_m &:= L_1 \cdot (L_2 \cdot \ldots \cdot L_n \,\widetilde{\otimes}\, R_1 \cdot \ldots \cdot R_m) \bigcup \\
&\quad\ (L_1 \cup R_1) \cdot (L_2 \cdot \ldots \cdot L_n \,\widetilde{\otimes}\, R_2 \cdot \ldots \cdot R_m) \bigcup \\
&\quad\ R_1 \cdot (L_1 \cdot \ldots \cdot L_n \,\widetilde{\otimes}\, R_2 \cdot \ldots \cdot R_m)
\end{aligned}$$

ArrgmtMerge $\Downarrow$: $\mathit{LabelArrangement} \times \mathit{LabelArrangement} \longrightarrow \mathcal{P}(\mathit{LabelArrangement})$

$\mathcal{A}_1 \Downarrow \mathcal{A}_2 := \{\mathcal{A}_x = L_1 \cdot \ldots \cdot L_r \in \mathcal{A}_1 \otimes \mathcal{A}_2 \mid \mathcal{A}_x$ satisfies the following conditions :

(i) **[Neighboring]** $\forall \triangleright_i^c, \triangleleft_i^c \in \mathbb{F},\ \triangleright_i^c \in L_t \rightarrow \triangleleft_i^c \in L_{t+1}$

(ii) **[Consistency]** $\neg\exists \triangleright_i^c, \triangleleft_i^c, \triangleright_j^d, \triangleleft_j^d \in \mathbb{F}$ and $c \neq d$, **s.t.** $\{\triangleright_i^c, \triangleright_j^d\} \subset L_t \ \vee\ \{\triangleleft_i^c, \triangleleft_j^d\} \subset L_t$

(iii) **[Uniqueness]** $\neg\exists\, l \in \mathbb{F}$, **s.t.** $l \in L_i \wedge l \in L_j \wedge i \neq j$

(iv) **[Non-overlapping]** $\neg\exists \triangleright_n^X, \triangleleft_n^X, \triangleright_m^X, \triangleleft_m^X \in \mathbb{F}$ and $m \neq n$, **s.t.** $\triangleright_n^X \in L_i \wedge \triangleleft_n^X \in L_k \ \wedge\ \triangleright_m^X \in L_j \ \wedge\ i < j < k$

GetArrgmtFromWord : $\mathit{Word} \qquad \mathit{LabelArrangement}$

GetArrgmtFromWord$({}^{L}c\mathcal{W}_t)$ $:= L \cdot$ **GetArrgmtFromWord**$(\mathcal{W}_t)$

GetArrgmtFromWord$({}^{L}X\mathcal{W}_t)$ $:= L \cdot$ **GetArrgmtFromWord**$(\mathcal{W}_t)$

GetArrgmtFromWord$({}^{L}\varepsilon)$ $:= L$

ProjectArrgmtToVar $\downarrow$: $\mathit{LabelArrangement} \times \mathit{StringVariable} \longrightarrow \mathcal{P}(\mathit{LabelArrangement})$

$$\downarrow_X (L_1 \cdot \ldots \cdot L_n) := \{L_t \cdot \ldots \cdot L_{t+m} \mid \exists^{R_1}X^{R_2} \in \mathbb{F},\ \textbf{s.t.}\ R_1 \cap L_t \neq \phi \ \wedge\ R_2 \cap L_{t+m} \neq \phi\}$$

Fig. 6 Arrangement primitives

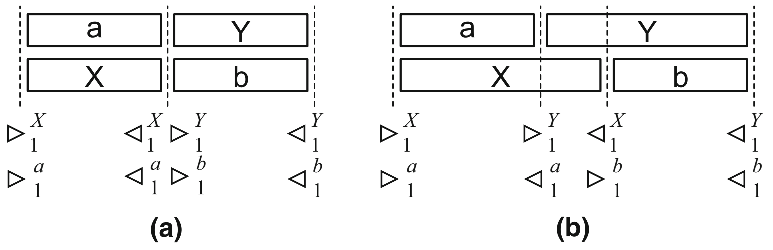


Fig. 7 Sample arrangements of $[\mathbb{E}_2]$

From this arrangement, we can easily derive two smaller equations $X = a$ and $Y = b$, which directly give a solution.

Arrangement production Given the LHS and RHS words of an equation, there are multiple (finite) ways to arrange their labels. For example, the following represents another arrangement of equation $[\mathbb{E}_2]$.

$$\left\{\rhd_1^a, \rhd_1^X\right\} \cdot \left\{\lhd_1^a, \rhd_1^Y\right\} \cdot \left\{\lhd_1^X, \rhd_1^b\right\} \cdot \left\{\lhd_1^Y, \lhd_1^b\right\} \qquad [\mathcal{A}_2]$$

It corresponds to the following graphical alignment shown in Fig. 7b. In the graphical representation, we can observe that the (left) boundary of Y "cut" the variable X into two parts which are hence constrained differently. We refer to this by saying that "Y cuts X". In fact, we can consider the labels of a word in the initial formula to denote an initial arrangement.

Operation **GetArrgmtFromWord**$(\mathcal{W})$ in Fig. 6 describes how the initial arrangement of $\mathcal{W}$ is recursively computed. It acquires the boundary label sets from left to right until the terminal symbol ε is encountered.

Next, we define the product operation $\otimes$ that produces a set of possible arrangements from two arrangements. Intuitively, one can understand that given the initial arrangements of the LHS and RHS words of an equation, the operator produces the possible label set arrangements, each denoting a possible alignment of the boundaries of the two words. According to the definition in Fig. 6, the operator first unions the two left-most label sets and the two right-most label sets, corresponding to aligning the word boundaries. Note that there is only one way to align word boundaries. It then uses a recursive operator $\widetilde{\otimes}$ to arrange the internal label sets. The definition of $\widetilde{\otimes}$ means that given the head label sets L_1 and R_1 in the two input arrangements, respectively, the head label set of the resulting arrangement can be L_1, R_1, or $L_1 \cup R_1$.

For our aforementioned equation $[\mathbb{E}_2]$, the product of the LHS and RHS word arrangements

$$\{\rhd_1^a\} \cdot \{\rhd_1^Y, \lhd_1^a\} \cdot \{\lhd_1^Y\} \;\otimes\; \{\rhd_1^X\} \cdot \{\rhd_1^b, \lhd_1^X\} \cdot \{\lhd_1^b\} = \{\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3\}$$

with $[\mathcal{A}_1]$ and $[\mathcal{A}_2]$ the arrangements defined earlier and $[\mathcal{A}_3]$ the following.

$$\{\rhd_1^a, \rhd_1^X\} \cdot \{\lhd_1^X, \rhd_1^b\} \cdot \{\lhd_1^a, \rhd_1^Y\} \cdot \{\lhd_1^Y, \lhd_1^b\} \qquad [\mathcal{A}_3]$$

For ease of presentation, we assume *none of the variables has the empty solution* ε. All the algorithms and theoretical results can be easily extended to handle ε solutions. Note that if variables X and Y in $[\mathbb{E}_2]$ could be ε, the $\otimes$ operator would produce more arrangements.

Valid arrangements Some of the arrangements produced by the $\otimes$ operator may not be valid, violating intrinsic properties in the string domain. We hence define a new operator $\Downarrow$, called *arrangement merging*, that filters out the invalid arrangements. According to the definition in

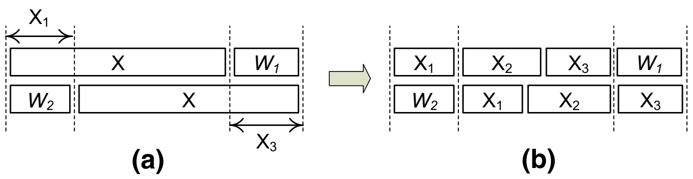


Fig. 8 The essence of overlapping arrangement. We omit the labels for readability

Fig. 6, the merge operator produces a subset of the arrangements generated by the $\otimes$ operator, pruning those that do not satisfy the four conditions listed:

- **[Neighboring]** condition states that the left boundary label and the right boundary label of a character need to be in the neighboring label sets. Intuitively, it states that a character cannot be cut/split by any boundary. As such, the arrangement $[\mathcal{A}_3]$ for our previous example is not valid as it suggests character a is cut and split by the right boundary of X.
- **[Consistency]** condition states that the left/right boundaries of different characters cannot be in the same label set. Intuitively, different characters cannot align.
- **[Uniqueness]** condition specifies that a label can only appear once in an arrangement. Recall that multiple occurrences of the same variable/character have different labels.
- **[Non-Overlapping]** condition is more complex and is discussed below.

An arrangement satisfying the first three conditions is also called a *valid* arrangement.
Non-overlapping arrangements We have discussed in Sect. 6.1, the formula reduction process may get into an infinite loop. That is, the reduced equations (for the sub-parts of the original equations) may have the same form of the original equations such that the reduction never ends. While we will formally discuss the theorem in Sect. 6, the intuition can be explained by Fig. 8. Figure (a) shows a boundary alignment of equation $X \cdot \mathcal{W}_1 = \mathcal{W}_2 \cdot X$. The two instances of X overlap and hence variable X is cut by two boundaries, one is $\lhd_1^X$ (i.e. the RHS label of the X on the left) and the other is $\rhd_2^X$. The two cuts may be at different positions inside X. As such, in the next step, X is split to $X_1 \cdot X_2 \cdot X_3$ with $X_1 = \mathcal{W}_2$ and $X_3 = \mathcal{W}_1$. Figure (b) shows the new alignment after the split. Note that the reduction yields an equation $X_2 \cdot X_3 = X_1 \cdot X_2$, which is essentially an re-occurrence of the form $X \cdot \mathcal{W}_1 = \mathcal{W}_2 \cdot X$, leading to infinite reduction.

As discussed in [41], overlapping solutions rarely happen in practice, especially in the context of program related string analysis. One key design choice is hence to detect and prune overlapping arrangements such that the procedure can avoid such non-terminations. Condition **[Non-overlapping]** in the definition of $\Downarrow$ in Fig. 6 defines the non-overlapping requirement.

Observe that the merge operation $\Downarrow$ generates a set of arrangements from two input arrangements. Our procedure selects one to proceed. If the selection does not lead to a SAT result, it is rolled back and a different selection will be taken. If none of the selections leads to SAT, the procedure returns UNSAT or indicates the presence of overlapping arrangements that have been pruned by the procedure. More details can be found in Sect. 6.2.3.
Per variable arrangement A variable may appear multiple times in multiple equations. The label arrangements of multiple occurrences of the variable may suggest the variable being cut differently and divided into different sub-parts. Such information essentially denotes the constraints on the variable from multiple equations. For example, if a variable X aligns with a character 'a' on its left boundary in one equation and aligns with a character 'b' on its right boundary in another equation, we know that X must start with 'a' and end with 'b'.

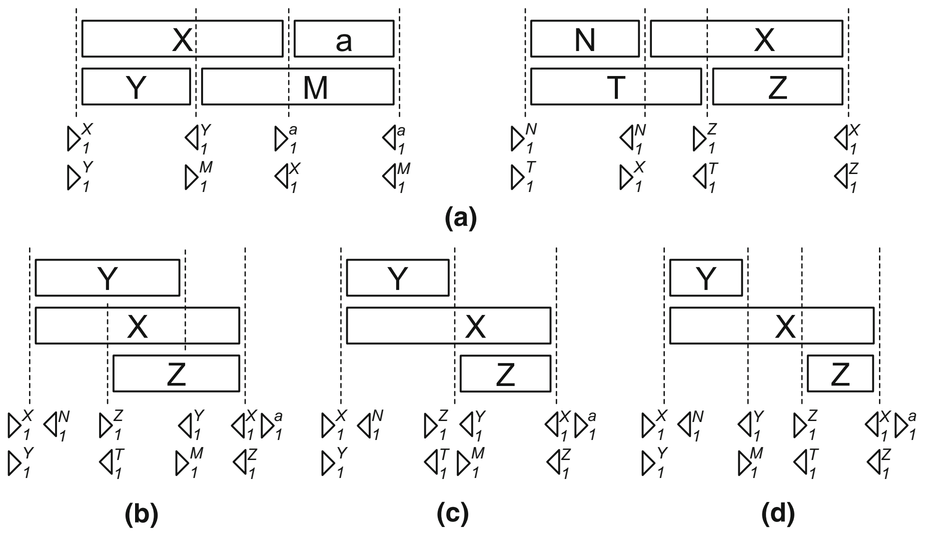


Fig. 9 Example for variable arrangement

In general, the constraints for the same variable from multiple equations can be very complex, requiring proper handling. For example, consider the following formula (the initial labels can be inferred and hence omitted for readability).

$$X \cdot a = Y \cdot M \ \wedge \ N \cdot X = T \cdot Z \qquad [\mathbb{E}_3]$$

Note that X aligns with Y on their left boundaries in the first equation and aligns with variable Z on their right boundaries in the second equation. We need to consider the relative positions of Y and Z inside X because they lead to different constraints of the sub-parts of X.

Figure 9a shows a valid arrangement for $[\mathbb{E}_3]$, which is generated by the aforementioned step of merging the LHS and RHS words of the equations. Observe that in this arrangement, X is cut by both Y and Z. Figures (b), (c), and (d) depict the three different relative positions of Y and Z regarding X. Note that they represent different constraints such that we need to generate different (smaller) equations in formula reduction. Particularly, in figure (b), Y and Z overlap such that they constrain each other while both constrain part of X. In (c), Y and Z covers X but they don't overlap. In (d), Y and Z constrain the head and the tail of X, respectively. But the middle part of X is not constrained by either.

Another novel aspect of our procedure is the fusion of the different constraints for the same variable from various equations. The observation is that *if we denote the constraint for a given variable from an equation as an arrangement, the variable arrangements from different equations can be merged using the merge operation* $\Downarrow$.

In Fig. 6, we define a variable arrangement projection operator: $\downarrow_X$ $(\mathcal{A})$ acquires the parts of the arrangement $\mathcal{A}$ related to variable X, that is, the set of subsequences of label sets that start with the LHS label of X and terminate with the corresponding RHS label of X. They contain all the boundaries that cut X in the given arrangement.

For example, let the arrangement of the equation on the left of Fig. 9 (a) be $\mathcal{A}_{a1}$, $\downarrow_X$ $(\mathcal{A}_{a1}) = \{\{\rhd_1^X, \rhd_1^Y\} \ \cdot \ \{\lhd_1^Y, \rhd_1^M\} \ \cdot \ \{\lhd_1^X, \rhd_1^a\}\}$. In the definition of $\downarrow$ in Fig. 6, given an arrangement $L_1 \cdot \cdots \cdot L_n$ and a variable X, it identifies all the subsequences $L_t \cdot \cdots \cdot L_{t+m}$,

REWRITEWD : $Word \times (StringVariable \rightarrow LabelArrangement) \qquad Word$

$$\text{REWRITEWD}({}^{L_1}c^{L_2} \cdot \mathcal{W}, \mathcal{A}_{var}[]) := {}^{L_1}c^{L_2} \cdot \text{REWRITEWD}({}^{L_2}\mathcal{W}, \mathcal{A}_{var}[])$$

$$\text{REWRITEWD}({}^{L_l}X^{L_r} \cdot \mathcal{W}, \mathcal{A}_{var}[]) := {}^{L_l \cup L_1}X_1^{L_2}X_2 \ldots {}^{L_{n-1}}X_{n-1}^{L_n \cup L_r} \cdot \text{REWRITEWD}({}^{L_n \cup L_r}\mathcal{W},\ \mathcal{A}_{var}[]),$$
in which $\mathcal{A}_{var}[X] = L_1 \cdot \ldots \cdot L_n$

$$\text{REWRITEWD}({}^{L}\varepsilon, \mathcal{A}_{var}[]) := {}^{L}\varepsilon$$

SPLITEQ : $Equation \times LabelArrangement \times (StringVariable \rightarrow LabelArrangement) \qquad Formula$

$$\text{SPLITEQ}(\mathcal{W}_1 = \mathcal{W}_2,\ \mathcal{A}_{eq},\ \mathcal{A}_{var}[]) := \text{RSPLITEQ}(\text{REWRITEWD}(\mathcal{W}_1, \mathcal{A}_{var}) = \text{REWRITEWD}(\mathcal{W}_2,\ \mathcal{A}_{var}),\ \mathcal{A}_{eq})$$

$$\text{RSPLITEQ}(\ {}^{L_i}\mathcal{W}_1^{L_j} \cdot \mathcal{W}_2 =^{L_k} \mathcal{W}_3^{L_l} \cdot \mathcal{W}_4,\ L_1 \cdot L_2 \cdot \mathcal{A}) := ({}^{L_i}\mathcal{W}_1^{L_j} =^{L_k} \mathcal{W}_3^{L_l}) \wedge \text{RSPLITEQ}({}^{L_j}\mathcal{W}_2 =^{L_l} \mathcal{W}_4,\ L_2 \cdot \mathcal{A}),$$
with $(L_i \cap L_k \cap L_1 \neq \phi) \wedge (L_j \cap L_l \cap L_2 \neq \phi)$

SOLVABLE : $Formula \qquad Boolean$

$$\text{SOLVABLE}(\mathbb{F}) := \begin{cases} true & \text{iff } \forall \mathbb{E} \in \mathbb{F}, \mathbb{E} \text{ must have the form of ``} X = Y \text{'' or ``} X = c \text{''} \\ false & \text{otherwise} \end{cases}$$

Fig. 10 Formula transformation primitives

in which L_t share some common labels with the LHS label set of X in the formula and L_{t+m} share some common labels with the corresponding RHS label set of X.

Once we get the arrangements for all the occurrences of a variable, we can use the arrangement merge operator $\Downarrow$ to merge them to generate all the possible (joint) arrangements inside the variable. In the example in Fig. 9a, the projection of the arrangement of the equation on the right, $\mathcal{A}_{a2}$, yields $\downarrow_X (\mathcal{A}_{a2}) = \{\{\triangleright_1^X, \triangleleft_1^N\} \cdot \{\triangleleft_1^T, \triangleright_1^Z\} \cdot \{\triangleleft_1^X, \triangleleft_1^Z\}\}$. The operation $(\downarrow_X (\mathcal{A}_{a1})) \Downarrow (\downarrow_X (\mathcal{A}_{a2}))$ hence yields exactly the three arrangements in figures (b)–(d).

Again, the procedure selects one variable arrangement from the generated set to proceed. Backtracking is allowed.

6.2.2 Formula transformation

In the previous subsection, we discuss how to generate arrangements from equations and generate arrangements for variables from equation arrangements. Once a variable arrangement is selected, the constraints for its sub-parts (i.e. their alignments with other variables and characters) are determined. Therefore, in this subsection, we discuss how to split equations to represent the constraints on subparts.

The process consists of two steps.

- In the first step, a variable is split to a set of new variables according to the variable arrangement. Each equation is rewritten by replacing each variable with the split variables.
- In the second step, each equation is split to a set of new equations, each constraining a sub-part of the original equation. This process is guided by determining the common labels between the label sets of the LHS and RHS words of the equations generated in the first step.

The **REWRITEWD** operator in Fig. 10 is for the first step. It takes a word and a mapping from variables to their corresponding arrangements, and generates a new word with the split variables. The operator is defined recursively. If the head of the word is a character, the resulting word inherits the same character and its labels. The operator continues to rewrite the remainder of the word. If the head of the word is a variable X, it splits the variable to a set of $n - 1$ variables X_1, …, X_{n-1}, where n is the number of label sets in the variable arrangement of X. This implies that X has $n-1$ sub-parts, delimited by the label sets. Also, the new variables inherit both the label sets L_l and L_r from the original word and the label sets in the variable arrangement. These labels denote the boundaries that each new variable aligns with.

The **SPLITEQ** operator denotes the second step. It takes an equation, the corresponding equation arrangement, and the mapping for variable arrangements, and then generates a formula, which is the conjunction of a set of equations. It first rewrites the LHS and RHS words of the equation and then uses the **RSPLITEQ** operator to split the rewritten equation. The **RSPLITEQ** operator takes the rewritten equation and the arrangement of the *original* equation, and generates a conjunction of new equations. The splitting is directed by the label sets of the original equation arrangement and the rewritten equation. In particular, given the first two labels $L_1 \cdot L_2$ of the original equation arrangement, it looks for the label sets in the rewritten LHS and RHS words that share some common labels with L_1 and L_2. A new equation is generated to denote the equivalence of sub-parts delimited by the aforementioned label sets. The new equation is conjoined with the other new equations generated by applying the operator to the remainder of the equation.

Example Consider the example in Fig. 9a. Assume the variable arrangement for X is the one in Fig. 9b. In other words,

$$\mathcal{A}_{var}[X] = \{\rhd^X_1, \rhd^Y_1, \lhd^N_1\} \cdot \{\rhd^Z_1, \lhd^T_1\} \cdot \{\lhd^Y_1, \rhd^M_1\} \cdot \{\lhd^X_1, \lhd^Z_1, \rhd^a_1\}$$

The LHS word $X \cdot a$ of the first equation is rewritten as follows:

$$\begin{aligned}\mathcal{W}'_l &= \mathbf{REWRITEWD}({}^{\{\rhd^X_1\}}X^{\{\lhd^X_1, \rhd^a_1\}}a^{\{\lhd^a_1\}},\ \mathcal{A}_{var})\\ &= {}^{\{\rhd^X_1, \rhd^Y_1, \lhd^N_1\}}X_1^{\{\rhd^Z_1, \lhd^T_1\}}X_2^{\{\lhd^Y_1, \rhd^M_1\}}X_3^{\{\lhd^X_1, \lhd^Z_1, \rhd^a_1\}}a^{\{\lhd^a_1\}}\end{aligned}$$

Note that X is split to X_1, X_2, and X_3 due to the cuts from Z and Y. Similarly, the RHS word $Y \cdot M$ is rewritten as follows:

$$\begin{aligned}\mathcal{W}'_r &= \mathbf{REWRITEWD}({}^{\{\rhd^Y_1\}}Y^{\{\lhd^Y_1, \rhd^M_1\}}M^{\{\lhd^M_1\}},\ \mathcal{A}_{var})\\ &= {}^{\{\rhd^X_1, \rhd^Y_1, \lhd^N_1\}}Y^{\{\lhd^Y_1, \rhd^M_1\}}M_1^{\{\lhd^X_1, \rhd^a_1\}}M_2^{\{\lhd^a_1, \lhd^M_1\}}\end{aligned}$$

M is split to M_1 and M_2 due to the cut by X.

Therefore, we split the first equation as follows. Note that $\mathcal{A} = \{\rhd^X_1, \rhd^Y_1\} \cdot \{\lhd^Y_1, \rhd^M_1\} \cdot \{\lhd^X_1, \rhd^a_1\} \cdot \{\lhd^a_1, \lhd^M_1\}$ is the equation arrangement.

$$\begin{aligned} & \mathbf{SPLITEQ}(Xa = YM,\ \mathcal{A},\ \mathcal{A}_{var})\\ =\ & \mathbf{RSPLITEQ}(\mathcal{W}'_l = \mathcal{W}'_r,\ \mathcal{A})\\ =\ & ({}^{\{\rhd^X_1, \rhd^Y_1, \lhd^N_1\}}X_1^{\{\rhd^Z_1, \lhd^T_1\}}X_2^{\{\lhd^Y_1, \rhd^M_1\}} = {}^{\{\rhd^X_1, \rhd^Y_1, \lhd^N_1\}}Y^{\{\lhd^Y_1, \rhd^M_1\}})\\ & \quad \bigwedge \mathbf{RSPLITEQ}({}^{\cdots}X_3^{\cdots}a^{\cdots} = {}^{\cdots}M_1^{\cdots}M_2^{\cdots},\ \ldots)\\ =\ & \ldots\\ =\ & (X_1 \cdot X_2 = Y) \wedge (X_3 = M_1) \wedge (a = M_2) \qquad [\mathbb{E}_4]\end{aligned}$$

Note that we omit the labels in the last step for readability. The second equation is similarly split to the following.

$$(N = T_1) \wedge (T_2 = X_1) \wedge (X_2 \cdot X_3 = Z) \qquad [\mathbb{E}_5]$$

□

With the new set of equations, the procedure repeats the steps of (1) computing equation arrangement; (2) computing variable arrangement; (3) splitting formula. As such, the equations represent equivalence between smaller and smaller portions of the original words. The process terminates if the formula is in solvable form.

Solvable form As defined in Fig. 10, a formula (i.e. a conjunction of equations) is in solvable form if each equation is either an equivalence between a variable and a character, or an equivalence between two variables. For variables that are directly or indirectly (i.e. through other variables) equivalent to a character, their solution is the character. Our solving processing, namely the *[Consistency]* condition in arrangement production (Sect. 6.2.1), ensures that the same variable is not equivalent to different characters. Variables that are not equivalent to any character, directly or indirectly, are free variables such that we can assign any characters to them.

6.2.3 Algorithm

The procedure is summarized in Algorithm 1. It takes a formula over word equations $\mathcal{Q}_w$ and the corresponding integer linear arithmetic constraints $\mathcal{Q}_l$ over the length function, produces SAT, UNSAT or UNKNOWN results. UNKNOWN means that the algorithm has encountered overlapping arrangements and pruned those arrangements, even though it did not find any SAT solution.

The algorithm consists of three steps. In step one (lines 9–15), it generates the set of possible arrangements for each equation and selects one from the set and stores it to $\mathcal{A}_{eq}$ to proceed. Each invocation to the **SELECT**() function is similar to a choice point in model checking, allowing back-tracking and selecting another option.

In step two (lines 16–17), variable arrangements for each variable are computed based on the previously selected equation arrangements. Again, each variable may have multiple arrangements.

Lines 18–25 denote the third step, in which the formula is split based on the selected variable arrangements.

In lines 26–29, the algorithm checks an overlapping arrangement has ever detected and pruned. If so, the procedure returns UNKNOWN, otherwise, it returns UNSAT.

Essence of the procedure The essence of the algorithm is to discover all the boundaries that are correlated and search for a total order of them. We say boundaries are correlated if they cut the same variable. Step one in the algorithm orders the boundaries within the LHS and RHS words using the merge operator $\Downarrow$. However, this does not discover all the correlated boundaries because a variable may be constrained differently in other equations. Therefore, in step two, it collects the ordered boundaries within a variable from all its occurrences and generates all the possible orders of the boundaries. For example in Fig. 9a, from the equation arrangement (step one), we only know Y cuts X. Through step two, we further know Z also cuts X and hence need to order the two cuts.

However, this does not expose all related boundaries. For example, suppose an additional equation $Z = J \cdot K$ is added to the formula. After the first round of the algorithm, $[\mathbb{E}_5]$ becomes

$$(N = T_1) \wedge (T_2 = X_1) \wedge (\cdots X_2^{\{\triangleleft_1^Y, \triangleright_1^M\}} X_3 \cdots = \cdots Z_1^{\{\triangleleft_1^J, \triangleright_1^K\}} Z_2 \cdots) \qquad [\mathbb{E}_6]$$

In other words, the boundary between J and K cuts Z and hence cuts X. But it is not identified as correlated to the other boundaries that cut X in the first round because the cut is indirect (through Z). Fortunately, the correlation will be exposed and the boundaries will be

Algorithm 1 A high-level description of Z3str2's Algorithm

```
Input: Word equations $\mathcal{Q}_w$, and the corresponding integer linear arithmetic constraints $\mathcal{Q}_l$ over the length function
Output: SAT / UNSAT / UNKNOWN
Definition: $\mathcal{AS}_{eq}/\mathcal{AS}_{var}$: mappings from equation/variable to set of label arrangements;
            $\mathcal{A}_{eq}/\mathcal{A}_{var}$      : mappings from equation/variable to arrangement.
1: procedure SOLVESTRINGCONSTRAINT($\mathcal{Q}_w$, $\mathcal{Q}_l$)
2:   if equations in $\mathcal{Q}_w$ are all in solvable form then
3:     if $\mathcal{Q}_w$ is UNSAT or $\mathcal{Q}_l$ is UNSAT then
4:       return UNSAT
5:     if $\mathcal{Q}_w$ and $\mathcal{Q}_l$ can be consistently determined as SAT together then
6:       return SAT
7:   Convert $\mathcal{Q}_w$ equisatisfiably in disjunctive normal form (DNF) formula $\mathcal{Q}_a$
8:   for each disjunct $D$ in $\mathcal{Q}_a$ do
       /* Convert each equation in $D$ to arrangements consistent with the length constraints */
9:     for each equation $\mathbb{E} : \mathcal{W}_1 = \mathcal{W}_2 \in D$ do
10:      $\mathcal{AS}_{eq}[\mathbb{E}] \leftarrow$ GETARRGMTFROMWORD($\mathcal{W}_1$) $\Downarrow$ GETARRGMTFROMWORD($\mathcal{W}_2$)
11:      for $\mathcal{A}$ in $\mathcal{AS}_{eq}[\mathbb{E}]$ do
12:        Extract the length constraint implied by $\mathcal{A}$.
13:        if $\mathcal{A}$ has inconsistent length constraint then
14:          Remove $\mathcal{A}$ from $\mathcal{AS}_{eq}[\mathbb{E}]$
         /* SELECT(): Similar to a choice point in model checking,
                  allowing backtracking and selecting another option.*/
15:      $\mathcal{A}_{eq}[\mathbb{E}] \leftarrow$ SELECT($\mathcal{AS}[\mathbb{E}]$)
16:    for each string variable $X$ do
         /* Merge per-equation arrangements to a set of possible global arrangements $\mathcal{AS}_{var}[X]$ */
         /* Operator $\Downarrow$ detects and prunes arrangements with overlaps from $\mathcal{AS}_{var}[X]$*/
17:      $\mathcal{AS}_{var}[X] \leftarrow (\downarrow_X (\mathcal{A}_{eq}[\mathbb{E}_1])) \Downarrow ... \Downarrow (\downarrow_X (\mathcal{A}_{eq}[\mathbb{E}_n]))$
18:    for each global arrangement combination selected from $\mathcal{AS}_{var}[X]$, ..., for all variables $X$, ... do
19:      Let $D ::= E_1 \wedge ... \wedge E_m$
         /* Split each variable to sub-variables according to the selected global arrangement */
20:      $D' \leftarrow$ SPLITEQ($E_1, \mathcal{A}_{eq}[E_1], \mathcal{A}_{var}$) $\bigwedge ... \bigwedge$ SPLITEQ($E_m, \mathcal{A}_{eq}[E_m], \mathcal{A}_{var}$)
21:      Convert $\mathcal{Q}_w$ equisatisfiably to a system $\mathcal{Q}'_w$ of simpler equation based on $D'$
22:      $\mathcal{Q}'_l$ is the corresponding new set of length constraints
23:      $r$ = SOLVESTRINGCONSTRAINT($\mathcal{Q}'_w$,$\mathcal{Q}'_l$)
24:      if $r \equiv$ SAT then
25:        return SAT
26:  if overlapping variables have ever been detected then
27:    return UNKNOWN
28:  else
29:    return UNSAT
```

ordered in the second round of the algorithm when [$\mathbb{E}_6$] is reduced. The algorithm repeats until all correlations are exposed and correlated boundaries are ordered.

Multiple layer overlaps In Sect. 6.1, we present an example of infinite reduction caused by a multiple layer loop, that is, the loop contains multiple rounds of reduction. This is much more difficult to detect and prevent compared to direct looping such as $aX = Xb$. However, we observe the essence of such looping is still overlapping arrangement (Sect. 6.2.1) compared to direct looping. The difference is that correlated boundaries are gradually exposed such that the boundary labels that overlap may only manifest themselves after a few rounds of reduction. Our algorithm can detect them through the **[Non-overlapping]** condition in the merge operator $\Downarrow$.

Example We revisit the third example in Sect. 6.1.

$$X \cdot M = N \cdot Y \quad \wedge \quad Y \cdot K = L \cdot X$$

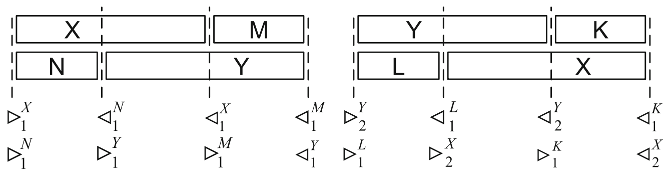

X
M
N
Y
Y
K
L
X
$\triangleright_1^X$ $\triangleleft_1^N$ $\triangleleft_1^X$ $\triangleleft_1^M$ $\triangleright_2^Y$ $\triangleleft_1^L$ $\triangleleft_2^Y$ $\triangleleft_1^K$
$\triangleright_1^N$ $\triangleright_1^Y$ $\triangleright_1^M$ $\triangleleft_1^Y$ $\triangleright_1^L$ $\triangleright_2^X$ $\triangleright_1^K$ $\triangleleft_2^X$

Fig. 11 The graphic representations of $\mathcal{A}_4$ and $\mathcal{A}_5$

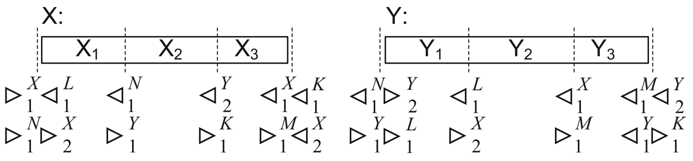

X:
X_1 X_2 X_3
Y:
Y_1 Y_2 Y_3
$\triangleright_1^X\triangleleft_1^L$ $\triangleleft_1^N$ $\triangleleft_2^Y$ $\triangleleft_1^X\triangleleft_1^K$ $\triangleleft_1^N\triangleright_2^Y$ $\triangleleft_1^L$ $\triangleleft_1^X$ $\triangleleft_1^M\triangleleft_2^Y$
$\triangleright_1^N\triangleright_2^X$ $\triangleright_1^Y$ $\triangleright_1^K$ $\triangleright_1^M\triangleleft_2^X$ $\triangleright_1^Y\triangleright_1^L$ $\triangleright_2^X$ $\triangleright_1^M$ $\triangleleft_1^Y\triangleright_1^K$

Fig. 12 The graphic representations of $\mathcal{A}_{var}[X]$ and $\mathcal{A}_{var}[Y]$

The equations with labels are

$$ {}^{\{\triangleright_1^X\}}X^{\{\triangleleft_1^X,\ \triangleright_1^M\}}M^{\{\triangleleft_1^M\}} = {}^{\{\triangleright_1^N\}}N^{\{\triangleleft_1^N,\ \triangleright_1^Y\}}Y^{\{\triangleleft_1^Y\}} \qquad [\mathbb{E}_7]$$
$$ {}^{\{\triangleright_2^Y\}}Y^{\{\triangleleft_2^Y,\ \triangleright_1^K\}}K^{\{\triangleleft_1^K\}} = {}^{\{\triangleright_1^L\}}L^{\{\triangleleft_1^L,\ \triangleright_2^X\}}X^{\{\triangleleft_2^X\}} \qquad [\mathbb{E}_8]$$

One arrangement of the LHS and RHS words of $[\mathbb{E}_7]$ and $[\mathbb{E}_8]$ are $[\mathcal{A}_4]$ and $[\mathcal{A}_5]$ respectively. Their graphic representations are shown in Fig. 11.

$$\{\triangleright_1^X,\ \triangleright_1^N\}\cdot\{\triangleleft_1^N,\ \triangleright_1^Y\}\cdot\{\triangleleft_1^X,\ \triangleright_1^M\}\cdot\{\triangleleft_1^M,\ \triangleleft_1^Y\} \qquad [\mathcal{A}_4]$$
$$\{\triangleright_2^Y,\ \triangleright_1^L\}\cdot\{\triangleleft_1^L,\ \triangleright_2^X\}\cdot\{\triangleleft_2^Y,\ \triangleright_1^K\}\cdot\{\triangleleft_1^K,\ \triangleleft_2^X\} \qquad [\mathcal{A}_5]$$

Now we apply the variable arrangement projection operator $\downarrow$ and get the arrangements related to variable X and Y.

$$\downarrow_X(\mathcal{A}_4) = \{\mathcal{A}_7\},\ \ \mathcal{A}_7 = \{\triangleright_1^X,\ \triangleright_1^N\}\cdot\{\triangleleft_1^N,\ \triangleright_1^Y\}\cdot\{\triangleleft_1^X,\ \triangleright_1^M\}$$
$$\downarrow_X(\mathcal{A}_5) = \{\mathcal{A}_8\},\ \ \mathcal{A}_8 = \{\triangleleft_1^L,\ \triangleright_2^X\}\cdot\{\triangleleft_2^Y,\ \triangleright_1^K\}\cdot\{\triangleleft_1^K,\ \triangleleft_2^X\}$$
$$\downarrow_Y(\mathcal{A}_4) = \{\mathcal{A}_9\},\ \ \mathcal{A}_9 = \{\triangleleft_1^N,\ \triangleright_1^Y\}\cdot\{\triangleleft_1^X,\ \triangleright_1^M\}\cdot\{\triangleleft_1^M,\ \triangleleft_1^Y\}$$
$$\downarrow_Y(\mathcal{A}_5) = \{\mathcal{A}_{10}\},\ \ \mathcal{A}_{10} = \{\triangleright_2^Y,\ \triangleright_1^L\}\cdot\{\triangleleft_1^L,\ \triangleright_2^X\}\cdot\{\triangleleft_2^Y,\ \triangleright_1^K\}$$

Then we can use the merge operator ($\Downarrow$) to generate the arrangements in variables. One of the arrangement combination generated by $\mathcal{A}_7 \Downarrow \mathcal{A}_8$ and $\mathcal{A}_9 \Downarrow \mathcal{A}_{10}$ is as follows. Figure 12 shows the corresponding graphic representations.

$$\mathcal{A}_{var}[X] = \{\triangleright_1^X, \triangleright_1^N, \triangleleft_1^L, \triangleright_2^X\}\cdot\{\triangleleft_1^N, \triangleright_1^Y\}\cdot\{\triangleleft_2^Y, \triangleright_1^K\}\cdot\{\triangleleft_1^X, \triangleright_1^M, \triangleleft_1^K, \triangleleft_2^X\}$$
$$\mathcal{A}_{var}[Y] = \{\triangleleft_1^N, \triangleright_1^Y, \triangleright_2^Y, \triangleright_1^L\}\cdot\{\triangleleft_1^L, \triangleright_2^X\}\cdot\{\triangleleft_1^X, \triangleright_1^M\}\cdot\{\triangleleft_1^M, \triangleleft_1^Y, \triangleleft_2^Y, \triangleright_1^K\}$$

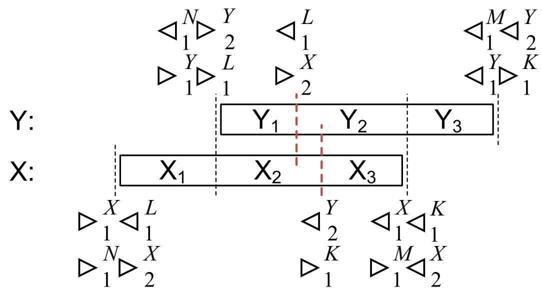
The LHS word $X\cdot M$ and RHS word $N\cdot Y$ can be rewritten as follows.

$$\begin{aligned}\mathcal{W}_l^{\mathbb{E}_7} &= \textbf{REWRITEWD}({}^{\{\triangleright_1^X\}}X^{\{\triangleleft_1^X,\triangleright_1^M\}}M^{\{\triangleleft_1^M\}},\ \mathcal{A}_{var})\\ &= {}^{\{\triangleright_1^X,\triangleright_1^N,\triangleleft_1^L,\triangleright_2^X\}}X_1^{\{\triangleleft_1^N,\triangleright_1^Y\}}X_2^{\{\triangleleft_2^Y,\triangleright_1^K\}}X_3^{\{\triangleleft_1^X,\triangleright_1^M,\triangleleft_1^K,\triangleleft_2^X\}}M^{\{\triangleleft_1^M\}}\\ \mathcal{W}_r^{\mathbb{E}_7} &= \textbf{REWRITEWD}({}^{\{\triangleright_1^N\}}N^{\{\triangleleft_1^N,\triangleright_1^Y\}}Y^{\{\triangleleft_1^Y\}},\ \mathcal{A}_{var})\\ &= {}^{\{\triangleright_1^N\}}N^{\{\triangleleft_1^N,\triangleright_1^Y,\triangleright_2^Y,\triangleright_1^L\}}Y_1^{\{\triangleleft_1^L,\triangleright_2^X\}}Y_2^{\{\triangleleft_1^X,\triangleright_1^M\}}Y_3^{\{\triangleleft_1^M,\triangleleft_1^Y,\triangleleft_2^Y,\triangleright_1^K\}}\end{aligned}$$

The equation $[\mathbb{E}_7]$ can be split as what is showed in Fig. 13.

$$
\begin{aligned}
& \textsc{split}\mathbf{EQ}(XM = NY,\ \mathcal{A}_4,\ \mathcal{A}_{var}) \\
= \ & \textsc{rsplit}\mathbf{EQ}(\mathcal{W}_l^{\mathbb{E}_7} = \mathcal{W}_r^{\mathbb{E}_7},\ \mathcal{A}_4) \\
= \ & ({}^{\{\triangleright_1^X, \triangleright_1^N, \triangleleft_1^L, \triangleright_2^X\}} X_1^{\{\triangleleft_1^N, \triangleright_1^Y\}} ={}^{\{\triangleright_1^N\}} N^{\{\triangleleft_1^N, \triangleright_1^Y, \triangleright_2^Y, \triangleright_1^L\}}) \\
& \bigwedge \textsc{rsplit}\mathbf{EQ}(\cdots X_2 \cdots X_3 \cdots M \cdots = \cdots Y_1 \cdots Y_2 \cdots Y_3 \cdots, \ldots) \\
= \ & \ldots \\
= \ & (\cdots X_1 \cdots = \cdots N \cdots) \wedge (\cdots M \cdots = \cdots Y_3 \cdots) \wedge \\
& \boxed{({}^{\{\triangleleft_1^N, \triangleright_1^Y\}} X_2^{\{\triangleleft_2^Y, \triangleright_1^K\}} X_3^{\{\triangleleft_1^X, \triangleright_1^M, \triangleleft_1^K, \triangleleft_2^X\}} ={}^{\{\triangleleft_1^N, \triangleright_1^Y, \triangleright_2^Y, \triangleright_1^L\}} Y_1^{\{\triangleleft_1^L, \triangleright_2^X\}} Y_2^{\{\triangleleft_1^X, \triangleright_1^M\}})} \qquad [\mathbb{E}_9]
\end{aligned}
$$

Fig. 13 Splitting equation $[\mathbb{E}_6]$



$\triangleleft_1^N \triangleright_2^Y$
$\triangleright_1^Y \triangleright_1^L$
$\triangleleft_1^L$
$\triangleright_2^X$
$\triangleleft_1^M \triangleleft_2^Y$
$\triangleleft_1^Y \triangleright_1^K$
Y:
Y_1
Y_2
Y_3
X:
X_1
X_2
X_3
$\triangleright_1^X \triangleleft_1^L$
$\triangleright_1^N \triangleright_2^X$
$\triangleleft_2^Y$
$\triangleright_1^K$
$\triangleleft_1^X \triangleleft_1^K$
$\triangleright_1^M \triangleleft_2^X$

Fig. 14 The graphic representations of $\mathcal{A}_{var}[X]$

For equation $[\mathbb{E}_9]$, we can get its arrangements if we repeat the same procedure for $[\mathbb{E}_7]$ and $[\mathbb{E}_8]$. Among them, an interesting one is the following:

$$\{\triangleright_1^Y, \triangleleft_1^N, \triangleright_2^Y, \triangleright_1^L\} \cdot \{\triangleleft_1^L, \triangleright_2^X\} \cdot \{\triangleleft_2^Y, \triangleright_1^K\} \cdot \{\triangleleft_1^X, \triangleright_1^M, \triangleleft_2^X, \triangleleft_1^K\} \qquad [\mathcal{A}_{11}]$$

We further compute the arrangement projection to variable X. As shown in Fig. 14, intuitively, the variable project is to determine the order between the boundaries in red, which are created in the previous cycle.

If we project these arrangements back to variable X, we get $\mathcal{A}_{var}[X]$:

$$\left\{\boxed{\triangleright_1^X}, \triangleright_1^N, \triangleleft_1^L, \triangleright_2^X\right\} \cdot \left\{\triangleright_1^Y, \triangleleft_1^N, \triangleright_2^Y, \triangleright_1^L\right\} \cdot \left\{\triangleleft_1^L, \boxed{\triangleright_2^X}\right\} \cdot \left\{\triangleleft_2^Y, \triangleright_1^K\right\} \cdot \left\{\boxed{\triangleleft_1^X}, \triangleright_1^M, \triangleleft_2^X, \triangleleft_1^K\right\}$$

The boxed labels violate the **[Non-overlapping]** condition. A loop is detected.

6.2.4 Soundness of the Z3str2 algorithm

In this section we sketch the soundness proof of Z3str2's algorithm given in Algorithm 1. For a detailed formal analysis we refer the reader to the associated tech-report. The soundness property of any decision procedure in an SMT solving context can be stated as "If the procedure returns UNSAT, then input formula is indeed UNSAT".

Theorem 1 *Algorithm 1 is sound, i.e., when Algorithm 1 reports UNSAT, the input constraint is indeed UNSAT.*

Proof To see that Z3str2 is sound, we show that the UNSAT returned at line 4 and line 29 are both sound. First observe that line 4 returns an UNSAT if either string or integer constraints are determined to be UNSAT. For string constraints, we use the algorithms described in [14] to decide the satisfiability of word (dis)equations in the solved form. The soundness of line 4 relies on the soundness of the procedure [14] and the integer solver (here Z3).

For the UNSAT returned at line 29, we show transformations impacting it are all satisfiability-preserving. If a transformation is satisfiability-preserving, it means its output formula is satisfiable if and only if its input formula is satisfiable. In particular, transformations at (i) line 7 (ii) line 10 (iii) line 17 and (iv) lines 20–21 are satisfiability-preserving: (i) The disjunctive normal form conversion at line 7 is obviously satisfiability-preserving. (ii) The conversion in line 10 is probably the most involved in terms of establishing soundness. This step is a variant of the idea of sound transformation of word equations to arrangements mentioned in Makanin's paper [27]. We can show that arrangement generation is satisfiability-preserving because each arrangement is a finite set of equations implied by the input system of equations. In addition, we extract length constraints from arrangements and they may conflict with the existing integer constraints. If so, we drop inconsistent arrangements based on the UNSAT results determined by the integer theory. Similarly, since we assume the integer theory is sound, this step is also satisfiability-preserving. (iii) At line 17, we systematically enumerate all feasible orders among boundary labels according to the definition of the arrangement merge (Fig. 6). This step is satisfiability-preserving. (iv) In lines 20 and 21, this step derives simpler equations by a satisfiability-preserving rewriting. Please see the technical report for proof details [40]. Note the REs are reduced to word equations so that they can be handled by this same procedure. In addition, although we prune arrangements at line 17, the answer can only be SAT or UNKNOWN once this happens. The algorithm is still sound. Therefore, we return UNSAT exactly when we can prove this to be the case.

7 String and integer theory integration

Basic length rules For strings X and Y, we assert the following: (1) $|X| \geq 0$ (2) $|X| = 0 \leftrightarrow X = \epsilon$ (3) $X = Y \rightarrow |X| = |Y|$ (4) $|X \cdot \cdots \cdot Y| = |X| + \cdots + |Y|$.

String and integer theory integration As discussed in Sect. 3, finding a consistent solution for both strings and integers can be expensive due to the infinite search space. The goal of string and integer theory integration is to achieve synergy from the two such that the procedure can converge faster. In particular, one theory will generate new assertions in the domain of the other theory, and vice versa. Inside the string theory, the set of arrangements that is explored is constrained by the assertions on string lengths, which are provided by the integer theory. On the other hand, the string theory will derive new length assertions when it makes progress in exploring new arrangements. These assertions are provided to the integer theory so that the search space is pruned.

Consider $X \cdot Y = M \cdot N$, where X, Y, M and N are nonempty string variables. It has three possible arrangements: $[a1]$ $X = M \cdot T_1 \wedge N = T_1 \cdot Y$; $[a2]$ $X = M \wedge N = Y$; $[a3]$ $M = X \cdot T_2 \wedge Y = T_2 \cdot N$. Assume the integer theory infers that $|X| > |M|$ or $|Y| < |N|$. Thus, only $[a1]$ is consistent with the length conditions. The string solver only needs to explore one arrangement instead of three. On the other hand, assume the string solver is exploring arrangement $[a1]$. It generates a new assertion $[a1] \rightarrow |X| = |M \cdot T_1| \wedge |N| = |T_1 \cdot Y| \wedge |X| > |M| \wedge |N| > |Y|$, which in turn triggers the Z3 core to add an integer assertion.

Note that different string solvers implement string and integer integrations in vastly different ways [8,26,36,39]. Bjørner et al. [8] focus on integration in a staged manner. Yu et al. [39] focus on integration via automata manipulations. CVC4 [26], S3 [36], and Z3str2 are integrations within the DPLL(T) architecture, where the algorithm only solves parts of the formula on demand and learns new constraints as it solves such that these implied constraints often cut the search dramatically. Compared to CVC4 and S3, our integration is tighter, powered by the bi-directional heuristics.

7.1 Search space pruning via binary search

In this subsection, we present a relatively new heuristic that is enabled by integration between the string and integer theories, and the ability for the string theory solver to use information directly from the integer solver, combine it with facts about strings, and generate new terms that guide the search. Although this work is preliminary and requires further evaluation, we have found it to be very promising and to demonstrate noticeable performance improvements.

We observed that Z3str2 runs slowly when the input constraints can only be satisfied by very long strings. As mentioned in Algorithm 1, when a consistent global arrangement is obtained, for each string variable that cannot be split further, Z3str2 tries to assign concrete values to it. This is a two-step procedure: Z3str2 first tries to fix its length and makes sure that no conflict will be triggered by this assignment. Next, Z3str2 proposes concrete values from a pool of all possible constant strings whose lengths meet the requirement obtained in the previous step.

Z3str2 already takes advantage of information from the integer theory to some extent when performing this search. However, it is possible to refine this information and use it to prune the search space much more effectively. In the following subsections, we discuss the issues related to the previous search strategy, and then present a new search heuristic that converges much more rapidly for cases where the previous strategy performs poorly.

7.1.1 Performance challenges imposed by solving long strings

The integer theory represents a variable's possible value by a range, and gradually refines this range during the search. After the problem's satisfiability is established, it starts to construct the model. In some sense, the integer theory decides variables' assignments lazily. However, this will block the string theory from making further progress: even though the string theory can access the integer theory via the bi-directional integration, usually it cannot obtain an assignment until a concrete length (instead of a range) has been fixed by the integer theory.

Therefore, the string theory in Z3str2 and integer theory in the Z3 core should work together to find consistent string lengths. Z3-str does so using a naive linear search approach. In particular, the string theory component continuously and incrementally queries the integer theory to get a suitable length for each variable through Z3 core.

For a string variable X to be assigned, Z3str2 linearly samples the search space and adds assertions for each value in the sample space. For this purpose, the first length axiom added is of the form $(|X| \geq i * k) \rightarrow (|X| = i * k) \vee \cdots \vee (|X| = (i+1) * k - 1) \vee (|X| \geq (i+1) * k)$, where k is a predefined non-negative integer representing the window size and i is a non-negative integer denoting the number of iteration. The Z3 core picks one assertion from the given set, verifies it against the existing context in the integer theory and backtracks if a conflict is detected. If $(|X| \geq (i+1) * k)$ is selected and included in the context, the string part goes to the next iteration and repeats the same procedure.

However, for constraints asserting large length, we found that the linear search consumes a significant amount of solving time in proposing gradually increasing length values, and such iterations impose unnecessary performance overhead on the whole decision procedure.

Consider the constraint $|X| \geq 52000$ that is derived from the current search context. Assume k is set to 4. In the first iteration, the length query added is $(|X| \geq 0) \rightarrow (|X| = 0) \vee \cdots \vee (|X| = 3) \vee (|X| \geq 4)$. Suppose initially the core picks the assertion $X = 2$ but it is not consistent with the length constraint. Therefore, the core backtracks and tries another assertion, for instance, $|X| \geq 4$. Then, the string theory asserts the next length axiom, $(|X| \geq 4) \rightarrow (|X| = 4) \vee \cdots \vee (|X| = 7) \vee (|X| \geq 8)$ and repeats the same procedure. In essence, the string plug-in needs to add the assertion in a linear order till it finds a valid one. The same process has to be done for all string variables, which significantly degrades the performance.

7.1.2 Binary search based heuristics

As discussed above, gradually increasing the length introduces redundant length query iterations as well as a large number of unnecessary length constraints for string variables. Therefore, we designed a new search space pruning technique based on the binary search with heuristic support for Z3 core's backtracking functionality.

However, as the strings are unbounded, we cannot fix an upper bound on their lengths. So, the traditional binary search based technique is not directly applicable. Instead, we design new heuristics, driven by binary search and backtracking, to guide the search. In particular, we chose a window with concrete lower and upper bound in which the size of the window varies dynamically in each iteration depending on the search criteria. We perform the binary search within the window while sliding the window from lower to higher values to find an upper bound. In other words, the lower and upper boundaries of the window are modified when the window slides. In addition, the window size expands or shrinks by the order of two based on the feedback provided by the integer theory. The backtracking ability of the heuristicshelps to guide the search and refine our choice of window.

Algorithm 2 Binary search based heuristics for string-integer solver

```
    Input: A string variable $X$.
    Output: The consistent assignment of the length of $X$ ($|X|$)
 1: procedure GETLENGTHASSIGNMENT($X, low, high$)
 2:     if $|X|$ has a concrete assignment $l$ in the integer theory then
 3:         return $l$
 4:     $mid = low + (high - low)/2$
 5:     assert an axiom: $(|X| < mid) \vee (|X| = mid) \vee (|X| > mid)$
 6:     query the integer theory and check the context
 7:     if $(|X| > mid)$ is selected by the Integer theory then
 8:         return GETLENGTHASSIGNMENT($X, mid + 1, high \times 2$)
 9:     else if $|X| = mid$ is selected by the Integer theory then
10:         return $mid$
11:     else
12:         return GETLENGTHASSIGNMENT($X, low, mid - 1$)
```

The binary search based heuristics for the theory of strings and integers is summarized in Algorithm 2. It is a recursive procedure driven by the feedback obtained from the integer theory. In turn, it also proposes possible ranges and advances the search inside the integer theory. In lines 2–3, it checks whether the length of X has already been assigned by the integer theory. If so, it directly returns the assignment found in the current context. Otherwise, it asserts an additional axiom and starts a modified version of binary search. It probes the context and adjusts the search window accordingly based on the feedback. In particular, it finds the middle element in the current window and asserts a length axiom at line 5. At line 6, the string theory checks the outcome of the axiom just asserted and prunes the search space in lines 7–12. Note that the higher bound is doubled if the lower half can be eliminated at line 8. This guarantees that the search can eventually cover a large string length in the unbounded search space.

8 Enhancing symbolic execution of C/C++ string manipulating programs

Improper string manipulations are an important cause of software defects, which make them a target for program analysis. A study [31] on concolic testing tools and their limitations shows that string operations make up a significant proportion of system-level code. The difficulty of analyzing these operations adds additional overhead to current symbolic execution engines.

In this section, we discuss how a constraint solver over string and integer theory can provide significant performance improvement in the symbolic execution of string manipulating programs written in C/C++. We compare the constraint expressiveness and the overall procedure among existing tools and our software to show the enhancement enabled by Z3str2.

8.1 Limitations introduced by modeling strings as bit-vectors

In current symbolic execution techniques, there is a semantic gap between the high-level notion of strings and low-level representation of program states and memory. Using a bit-vector solver is useful for low-level program analysis as it captures constraints with bit-level precision and efficiently reasons about arithmetic overflows and bitwise operations. Bit-vector solvers also allow for reasoning about symbolic memory regions.

Symbolic execution engines (e.g., KLEE [9] and S2E [10]) collect constraints as bit-vectors by accumulating branch predicates and solving for the branch conditions using constraint solvers such as STP [13] and Z3 [12]. However, these engines perform poorly on programs containing string functions as they don't capture the high-level semantics of strings in accordance with the low-level bit-vector representation of all program data [10].

In particular, current symbolic execution engines that track constraints at the bit-vector level suffer from the path explosion problem. One of the main reason is that representing strings with bit-vectors implies that string operations have to be modeled as loops that check string values on character level. For example, commonly used string functions, such as *strlen* or *strcmp*, mainly consist of loops that iterate over characters until either a particular condition is met or the current character is the null terminator, marking the end of the string. To do so, the symbolic execution engine usually forks a fresh state per each possible length, leading to path explosion, especially when the string is unbounded.

For example, an invocation of the string library function *strlen* on a symbolic string S of size N will generate a total of $N + 1$ paths, one for each possible value of the length between 0 and N, regardless of how the length of the string is actually used throughout the rest of the program. In essence, uninteresting regions of code are explored by enumerating each branch without efficiently pruning the search space. Prior works from the S2E [10] group explained these performance bottlenecks with strings in their symbolic analysis. These restrictions could be lifted by applying string solvers that reason about strings as a primitive data type, similar to integers and bit-vectors, in the context of symbolic execution.

Moreover, it is difficult to detect certain classes of security vulnerabilities arising from certain overflow and underflow errors with state-of-the-art symbolic execution engines. For

Fig. 15 Login method demonstrating an overflow vulnerability

```
bool check_login(char* username, char* password){
    unsigned short len = strlen(username)+1;
    if(len > 32){
        invalid_login_attempt();
        exit(-1);
    }
    char* _username= (char*) malloc(len);
    strcpy(_username, username);
    ...
}
```

instance, if we analyze the history of integer-related overflows in CVE database, we observe that operations on extremely long strings are one of the primary sources of integer overflow vulnerabilities. Since the strings have to be converted to bit-vector representations, it requires the symbolic execution engine to fix concrete string lengths well in advance of where the strings are actually used. Essentially, the engine has to enumerate a large number of possible string lengths without knowing if they are necessary from the point of view of the string theory. As a result, existing symbolic execution engines powered by bit-vector-based solvers are not able to reason efficiently about arithmetic overflow/underflow errors caused by improper handling of string values. This important class of security vulnerabilities inspired us to propose a new theory combination for the underlying solver which allows it to efficiently reason about arithmetic overflow/underflow errors and memory corruption caused by incorrect string operations.

We use a motivating example to explain the limitations of existing string solvers in the context of low-level program analysis and the importance of the string-bitvector theory combination. Consider the *check_login()* function shown in Fig. 15. Here, the program calculates the length of the user-controlled input value *username*, and adds 1 to accommodate the trailing *null* character. A new buffer is allocated for the resulting string and the program copies *username* into it using the *strcpy* function. This code behaves as intended for normal-sized input. However, an integer overflow occurs if the user submits a *username* that is 65,535 characters long. The variable *len* is declared as `unsigned short`, which is a 16-bit integer that can hold any value between 0 and 65,535. When a string of length 65,535 is submitted as *username*, the result of *strlen(username)*+1 overflows to 0. This results in allocation of a buffer of size 0 on the heap with the call to *malloc*(). Attempting to copy *username* into this buffer results in memory corruption and the potential for an exploit.

8.2 Enhancements enabled by string-integer solvers in symbolic execution

Compared to bit-vector-based string constraint solving, modeling strings as a primitive type and tight string-integer theory solver integration make constraint modeling more expressive, as well as eliminating a significant number of unnecessary path enumerations in the context of symbolic execution.

Language expressiveness Strings modeled as bit-vector arrays are inherently bounded, whereas the length of strings modeled as a primitive type can be arbitrarily (finitely) large. As a result of this, a bit-vector-based string solver cannot precisely encode the overflows caused by a limited-precision type. Due to the upper bound on string length. bit-vector-based encoding has difficulty reasoning about the overflow that occurs in the previous example.

Eliminating loops over characters Since the solver is able to handle strings of unbounded length and reason about string operations at a high level, it is possible to abstract away low-level loops in some string-handling methods. For example, we can completely abstract away loops over individual characters in the heavily-used *strcmp*() and *strlen*() functions, as the solver has high-level knowledge of the semantics of these methods. As a result, by eliminating the loops we also eliminate a large number of potential symbolic paths that would otherwise be explored.

9 Experimental results

In this section, we describe the implementation of Z3str2, as well as experiments to validate the efficacy of the first two new techniques proposed in this paper, namely, overlapping variable detection and deeper string/integer theory integration in Sect. 9.1. However, typical benchmarks used to evaluate search space pruning techniques rarely involve large strings. To demonstrate the effectiveness of the optimized search heuristics, in Sect. 9.2, we compare the performance of the binary search and the linear search heuristics using a set of benchmarks that systematically cover different string length settings. We also compare with other solvers on a set of constraints requiring large strings. Finally, we evaluate the enhancements in the context of symbolic execution enabled by a constraint solver over string and integer theories in Sect. 9.3.

9.1 Evaluating search space pruning techniques

Both techniques improve solver efficiency in isolation, as well as when used simultaneously. We report both combined and isolated contributions.

1. *Detection of overlapping variables* During solving, Z3str2 prunes away arrangements with overlapping variables, leading to a smaller search space. Thus, if the technique is effective, we would be able to observe that other solvers time out on the cases reported as UNKNOWN by Z3str2. In Z3str2, an UNKNOWN result is returned when at least one arrangement with overlapping variables is found (and pruned), and no satisfying solution can be found over all arrangements with non-overlapping variables.
2. *Evaluating string and integer theory integration* The contribution of string-integer theory integration will be illustrated by demonstrating improvement in the performance of solving both SAT and UNSAT instances, in comparison with other solvers.

9.1.1 Experimental setting

We compare Z3str2 against five state-of-the-art string solvers, namely, CVC4 [26], S3 [36], Kaluza [33], PISA-MONA [35], and Stranger [38] across four different suites of benchmarks obtained from Kudzu/Kaluza [33], PISA [35], AppScan Source [2] and Kausler's [22] projects. Given the rich and diversified landscape of string problems, we chose to validate our approach using benchmarks from real-world applications with different characteristics. Additionally, the total number of tests on which we compared Z3str2 with other solvers is approximately 69,000.

Table 1 summarizes the experimental settings. Z3str2 and CVC4 both accept constraints in similar formats, and so we run Z3str2 and CVC4 on all suites. PISA-MONA and Stranger can only be invoked by APIs. Although S3 and Kaluza accept input from a file, the input language they understand has limited expressiveness. Therefore, we compare Z3str2 to them using the benchmarks over which those solvers were originally evaluated.

Table 1 Comparison study settings

	Z3str2	Z3str2 w/o integration	CVC4	S3	Kaluza	PISA-MONA	Stranger
Kaluza suite	✓	✓	✓	✓	✓		
PISA suite	✓		✓			✓	
AppScan suite	✓		✓				
Kausler suite	✓		✓				✓

Kaluza benchmark suite The Kaluza constraints were generated by a JavaScript symbolic execution engine [33], where length, concatenation and (finite) RE membership queries occur frequently. Despite the redundancy across many of the problems, it is still a "classical" test suite over which string solvers are commonly benchmarked. Both CVC4 and S3 were originally evaluated only on this suite, which consists of approximately 50,000 problems in the Kaluza format. The CVC4 team selected 47,284 of them, and translated them into the CVC4 format. The S3 team did the translation to S3 format. We wrote translators from CVC4 to Z3str2, and from Z3str2 to CVC4. The timeout threshold for comparison over this suite was set at 20 s per problem, which was the threshold used in CVC4 [26].

PISA benchmark suite While the Kaluza suite is large and includes string problems of varying sizes, it only contains a small subset of string operations. To make the comparison more comprehensive, we included constraints from real-world Java sanitizer methods that were used in the evaluation of the PISA system [35]. Sanitizers cleanse user input to remove the threat of an injection attack. They are usually complex and utilize various primitive string operations. We generated two groups of constraints: First, as in the PISA paper, we encode the semantics of the sanitizers and check the return value(s) against predefined attack patterns (such as cross-site scripting (XSS)). In the second group, we also encode input constraints per the application defining the sanitizer. For the PISA suite, we set a timeout of 200 s due to its higher complexity.

AppScan benchmark suite The third suite of benchmarks is derived fr om security warnings output by IBM Security AppScan Source Edition [2], an application sold commercially by IBM. These reflect potentially vulnerable information flows, represented as traces of program statements, which yield more representative real-world constraints than focusing on sanitizers only. We ran AppScan on popular websites to obtain traces. Similar to the PISA benchmarks, the AppScan constraints also utilize a rich set of string operators. As with PISA, the timeout here was also set at 200 s per benchmark.

Kausler benchmark suite The final suite is extracted from eight Java programs by Scott Kausler [22]. They represent path conditions obtained from dynamic symbolic execution, and are pure string constraints [21]. Unlike other benchmarks, Kausler's suite does not dump string constraints to file but instead calls the solvers via an API. The suite contains 174 path condition encoding files, and the resulting constraints are input to the solvers in-memory via their APIs. The comparison [22] was originally done using a driver interface [3]. However, we observed bugs ranging from JNI issues for Stranger to generation of invalid constraints for Z3str2. We made our best attempt to compare Z3str2 with Stranger using modified interfaces [1] patched by both the Stranger team and us.

We performed all the experiments on a workstation running Ubuntu 12.04 with an Intel i7-3770 CPU and 8 GB of RAM memory. For reproducibility, we have made the Z3str2

Table 2 Results on Kaluza suite [33]

	Z3str2		Z3str2 w/o integration		CVC4		S3		Kaluza	
	$\surd$	$\times$	$\surd$	$\times$	$\surd$	$\times$	$\surd$	$\times$	$\surd$	$\times$
sat	34,859	0	32,752	0	33,190	0	32,503	488	21,651	n/a
unsat	11,799	0	11,313	0	11,625	0	11,351	412	12,099	10,909
unknown	626†		395†		0		0		0	
Timeout	0		2824		2469		989		340	
Tool reports error	0		0		0		2		2285	
Crash	0		0		0		1539		0	
Total time (s)	4288.8(**1x**)		61,232.8(**14.3x**)		52,478.8 (**12.2x**)		22,543.4 (**5.3x**)		46,753.9 (**10.9x**)	

† 'unknown' indicates Z3str2 detected and avoided overlapping arrangements

source code publicly available.[4] For the other solvers, we used V1.5-prerelease of CVC4; the version of S3 from the original paper [36]; the Kaluza version from the CVC4 paper with "var" as the query string; and Stranger from [4].

9.1.2 Performance results

Kaluza suite In Table 2, "tool reports error" counts the number of inputs on which the solver reports an error. "crash", instead, refers to runtime errors such as segfaults. For "sat" and "unsat", $\times$ denotes the number of provably incorrect results (either an "unsat" response where the problem has a verified solution or a "sat" response with an infeasible solution, as defined in [26]), and $\surd$ the rest. The comparison involves Z3str2 without bidirectional string-integer theory integration, CVC4 and S3, but not PISA, as PISA cannot model string lengths or symbolic arithmetic operations which are intensive in the suite.

According to Table 2, neither Z3str2 nor CVC4 report any provably incorrect result, though Z3str2 is more effective and can solve more cases (46,658 compared to 44,815) without timeouts. Though Z3str2 additionally has 626 unknown cases, CVC4 times out on all these cases. Z3str2 without bidirectional integration solves 2593 fewer cases and times out more often. S3 has errors in both directions, as well as a total of 989 timeouts, while Kaluza suffers from fewer timeouts (340) but has many sat-as-unsat errors (10,909). Kaluza therefore is unsound. Since Kaluza only provides assignments for variables matching the query, SAT answers are not verifiable. Both S3 and Kaluza also have tool errors (2 and 2285, respectively). In addition, S3 crashes on 1539 cases. To compare performance on the sat and unsat Kaluza cases across the different solvers, we created the cactus plots in Figs. 16a (sat) and 17a (unsat). The vertical axis measures solving time on a logarithmic scale; the horizontal axis counts the total number of instances that (individually) took at most the indicated time to be solved. Incorrect results are excluded. In both categories, Z3str2 and CVC4 have comparable performance, while Z3str2 solves more cases and is faster on complex cases. S3 and Z3str2 without string-integer integration are slower. Kaluza has the worst performance.

PISA suite Table 3 presents the results on the PISA benchmarks. The "string operators stats" column lists the involved operations and their number of occurrences. In addition, we also

[4] https://sites.google.com/site/z3strsolver/.

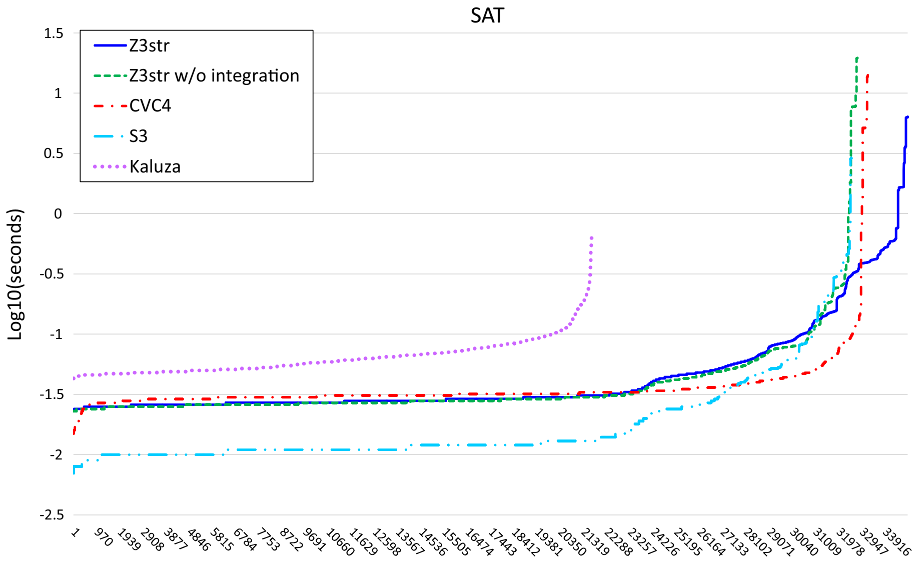


Fig. 16 Cactus plots for the Kaluza benchmark (w/o incorrect results): SAT instances

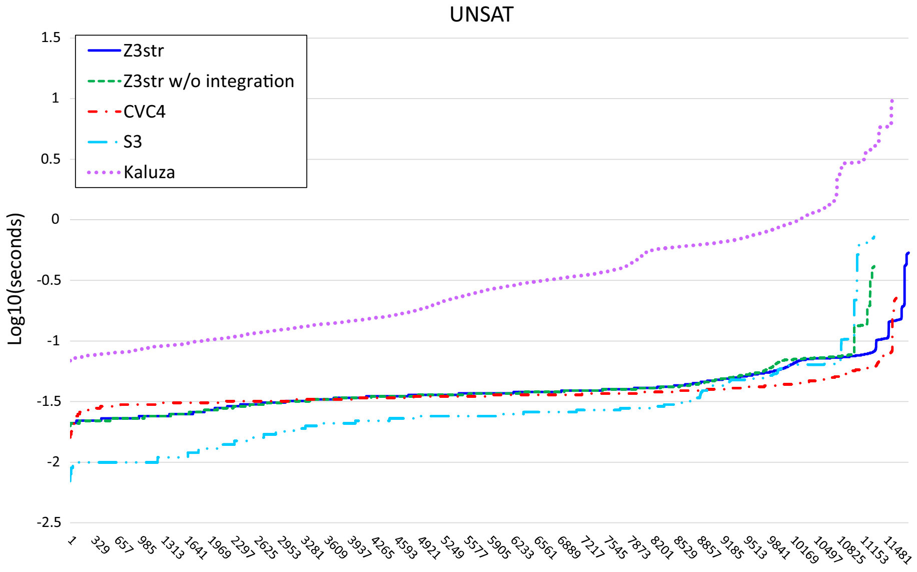


Fig. 17 Cactus plots for the Kaluza benchmark (w/o incorrect results): UNSAT instances

count the number of variables and predicates for each format. In this comparison, we included CVC4 and PISA, but not S3 and Kaluza, as we were not able to model popular string operations such as indexof using their language. Besides, for PISA, while one group of constraints is equivalent to the MONA program generated by PISA, enabling proper comparison, the other group requires changes to the PISA translation algorithm (to fix the input constraints as well as the negative output constraints), and thus the respective comparisons were not possible. From Table 3, we have the following observations. First, Z3str2 reports 8 sat cases compared to 6 by CVC4 and 2 timeouts. For the 6 sats in common, Z3str2 solves them in 1.069 s while CVC4 requires 51.394 s. Second, MONA and Z3str2 are in agreement. MONA runs faster on the sat cases, though it cannot generate satisfying string assignments, and has comparable performance to Z3str2 on the unsat cases.

AppScan suite The results of the third comparison over the AppScan benchmarks appear in Table 4. Correlation between variables is relatively high, as demonstrated by the "var" and "pred" columns. As shown in the table, Z3str2 reports sat on 8 cases while CVC4 agrees on 4 and times out on the rest. The performance gap between the solvers on sat cases in agreement is significant: Z3str2 completes in 0.707 s, whereas the CVC4 solving time is 154.852 s.

Kausler suite Table 5 shows the average solving time of the instances generated by the constraint generation tool in the Kausler suite. We were able to run the Stranger tool on five sets in this suite. Both CVC4 and Z3str2 are faster than Stranger on two of these sets, while Stranger is faster than Z3str2 and CVC4 on the remaining three.

However, these findings in the comparisons with Stranger should be qualified. First, Stranger crashes or hangs on 98 files. Z3str2 neither crashes nor hangs nor times out on any of the generated instances. We have omitted these 98 files from our comparison. Additionally, Stranger over-approximates disequalities ($\neq$operator) among variables that can represent multiple strings [5]. We observe that such cases commonly exist in all sets (the percentages of instances with $\neq$ operators in each set are 83.4, 61.7, 79.0, 96.0 and 95.0% respectively, and many fall into this category of disequalities among variables that represent multiple strings). This implies that Stranger produces unsound results. We believe that some of these constraints are easy for Stranger thanks to this over-approximation. By contrast Z3str2 correctly implements all operators and predicates in its input language. Finally, Stranger requires that integers occurring as indices and length bounds be constant, whereas Z3str2 and most other competing solvers support integers symbolically thus providing expressive power that is essential in practice.

The results also show that CVC4 and Z3str2 have comparable performance on three sets but CVC4 is faster than Z3str2 on the other two. A closer look at the constrains in *mathParser* and *naturalCLI* suggests that CVC4 usually has better performance when the variables are loosely constrained. We plan to introduce heuristics for such cases to improve Z3str2's performance in future versions.

The general trend, across all benchmark suites, is that CVC4 has comparable performance to Z3str2 although CVC4 times out far more often than Z3str2, whereas S3 is significantly slower. These results establish the efficacy of both techniques presented in this paper.

Detection of overlapping variables Z3str2 can decide either sat or unsat on 98.7, 100 and 100 of the instances in the Kaluza, PISA and AppScan suites, respectively. CVC4, in comparison, achieves 94.8, 50 and 50%. For unknowns reported by Z3str2 on the Kaluza instances, which occur in merely 1.3% of the cases, CVC4 times out on all of them. This lends support to our design choice of purposely pruning away parts of the solution space (those with overlapping arrangements) to avoid nontermination.

String and integer theory integration As the comparisons between Z3str2 versions with and without the integration clearly demonstrate, there is significant gain thanks to tightening the integer and string theory integration, which enables generation of implied constraints in both domains for more aggressive elimination of assignments unsatisfying for combined string-integer constraints.

Table 3 Results on constraints generated from sanitizers detected by PISA [35]

Input	String operators stats (omitting eq and dis-eq)	Z3str2				CVC4				PISA-MONA			
		var	pred	Result	Time (s)	var	pred	Result	Time (s)	var	pred	Result	Time (s)
pisa-000.smt2	contains (3), indexof (1), substring (1)	4	12	sat (√)	0.164	4	12	sat (√)	0.264	9	301	sat (?$^+$)	0.029
pisa-001.smt2	contains (1), indexof (1), substring (1)	4	9	sat (√)	0.114	4	9	sat (√)	0.032	–$^+$	–$^+$	–$^+$	–$^+$
pisa-002.smt2	contains (4)	2	10	sat (√)	0.114	2	10	sat (√)	50.871	–	–	–	–
pisa-003.smt2	contains (3), concat (1)	3	11	unsat (√)	0.064	3	11	timeout	200.00	–	–	–	–
pisa-004.smt2	contains (2), indexof (1), length (1), lastIndexof† (1), substring (2)	7	22	unsat (√)	0.038	10	32	timeout	200.00	9	331	unsat (√)	0.041
pisa-005.smt2	indexof (1), lastIndexof† (1), length (1), substring (2),	7	23	sat (√)	0.115	10	33	sat (√)	0.165	–	–	–	–
pisa-006.smt2	indexof (1), lastIndexof† (1), length (1), substring (2), contains (1)	7	24	unsat (√)	0.039	11	36	timeout	200.00	9	331	unsat (√)	0.038
pisa-007.smt2	indexof (2), lastIndexof†(1), length (1), substring (2), contains (1)	8	26	unsat (√)	0.042	11	36	timeout	200.00	9	324	unsat (√)	0.039
pisa-008.smt2	replace (5), contains (2)	6	13	sat (√)	0.214	6	13	timeout	200.00	9	283	sat (?$^+$)	0.031
pisa-009.smt2	replace (2), concat (1), contains (2)	3	8	sat (√)	0.447	3	8	sat (√)	0.046	9	292	sat (?$^+$)	0.054
pisa-010.smt2	replace (2), concat (1)	3	6	sat (√)	0.165	3	6	timeout	200.00	–	–	–	–
pisa-011.smt2	replace (1), concat (2)	3	6	sat (√)	0.115	3	6	sat (√)	0.016	–	–	–	–

+ We could not generate constraints without changing PISA. No string solutions are generated so it's not verifiable.
† CVC4 doesn't provide operator 'lastIndexof'. We encode it with operators "concat", "length" and "contains"

Table 4 Results on constraints derived from AppScan traces [2]

input	string operators stats (omitting eq and dis-eq)	Z3str2				CVC4			
		var	pred	result	time (s)	var	pred	result	time (s)
t01.smt2	indexof (4), substring (3)	7	37	sat (√)	0.265	7	37	timeout	200.00
t02.smt2	concat (3), membership (1), regexConcat (2), regexUnion (14), str2Regex (17), length (1)	5	47	sat (√)	0.215	5	33	sat (√)	0.026
t03.smt2	concat (3), membership (1), regexConcat (2), regexUnion(14), str2Regex (17), length(1)	5	46	sat (√)	2.519	5	32	timeout	200.00
t04.smt2	concat (5), membership (1), regexConcat (2), regexUnion (14), str2Regex (17), length(1)	6	50	sat (√)	4.574	6	35	timeout	200.00
t05.smt2	concat (3), membership (1), regexConcat (2), indexof (1) regexUnion (14), str2Regex (17), length (2), substring (1)	8	56	sat (√)	2.770	8	42	timeout	200.00
t06.smt2	concat (1), indexof (3), endsWith (5)	5	33	sat (√)	0.214	5	33	sat (√)	3.021
t07.smt2	concat (6), regexStar (2), str2Regex (4), endsWith (2) regexUnion (2), membership (2), startsWith (2)	8	32	sat (√)	0.114	8	29	sat (√)	0.115
t08.smt2	concat (2), regexStar (2), str2Regex (4), endsWith (2) regexUnion (2), membership (2), startsWith (2)	5	23	sat (√)	0.164	5	22	sat (√)	151.663

Table 5 Average solving time per constraint instance generated from Kausler suite

	Instance #	Stranger (ms)	CVC4 (ms)	Z3str2 (ms)
beasties	7230	51.8	6.4	6.4
jerichoHTMLParser	1275	5.9	9.5	10.7
mathParser	9138	1.4	12.8	39.9
mathQuizGame	21	9.4	8.5	7.1
naturalCLI	2367	3.0	10.0	23.4

9.2 Evaluating binary search based heuristics for large strings

The binary search based approach with backtracking functionality helps to prune the search space efficiently when the input contains operations on very long strings. We used the string-integer solver for the comparison and performance evaluation of the binary search approach. We performed two sets of experiments for this evaluation, which are discussed below.

9.2.1 Performance evaluation of string lengths in Z3str2

The first set of experiments demonstrates the inefficiency of the linear search heuristic in the previous version of Z3str2. For this purpose, we chose a simple constraint set containing only the length function. The constraints we chose are of the form $length_{str} = l$, where the choice of l is as described in Table 6. We performed a series of 11 tests and used 200 s as the timeout for each test case. The stable release of the Z3str2 solver in the linear search mode solved constraints of length up to 5000, but timed out on all larger instances. The solver took more than 12 h to terminate when the length is on the order of 10^5. However, the solver in the binary search mode performed very efficiently and solved all instances for l up to and including 10^6. A comparison of solving time for the two search heuristics in Z3str2 is shown in Table 6 and the cactus plots in Fig. 18.

Table 6 Performance of Z3str2 over string lengths

Benchmark	Length (l)	Binary search (s)	Naive search (s)
test1	10	0.042	0.040
test2	50	0.045	0.041
test3	100	0.039	0.051
test4	500	0.041	0.254
test5	1000	0.055	0.928
test6	5000	0.056	46.699
test7	10, 000	0.088	Timeout
test8	50, 000	0.323	Timeout
test9	100, 000	0.842	Timeout
test10	500, 000	21.084	Timeout
test11	1, 000, 000	105.636	Timeout

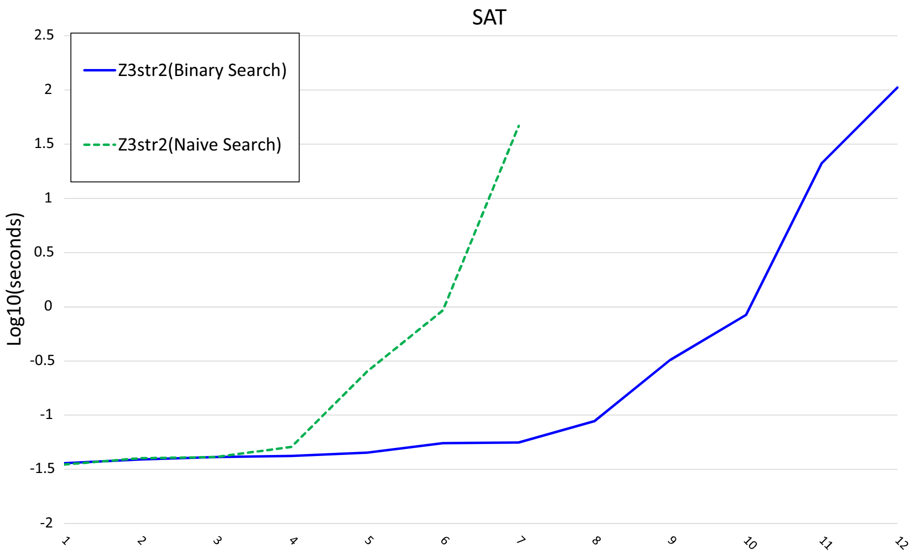


Fig. 18 Cactus plots for the length test

Table 7 Performance on benchmark suite

	Z3str2 (binary search)	Z3str2 (naive search)	CVC4
SAT	169	138	126
UNSAT	34	34	19
UNKNOWN	2	2	0
Timeout	0	31	60
Tool reports error	0	0	0
Crash	0	0	0
Total time (factor)	41.697 (**1x**)	9569.639 (**x229**)	12,014.893 (**x264**)

† ‘unknown’ indicates Z3str2 detected and avoided overlapping arrangements

9.2.2 Evaluation on benchmark suite

To measure the efficacy of our approach, we designed a benchmark suite containing 205 test cases, containing handcrafted constraints involving operations on large strings. The benchmark consists of constraints on various string operations supported by Z3str2 such as *Length, Concat, Indexof, Substring, EndsWith, StartsWith, Replace* , etc. Furthermore, we translated the constraints into the CVC4 input language and compared the performance of the benchmarks against CVC4 version 1.5. We used 200 s as the timeout for each constraint input.

The results of the comparison are presented in Table 7. The *SAT* and *UNSAT* column denotes the number of SAT and UNSAT answers respectively. According to Table 7, Z3str2 with binary search solves all test instances very quickly compared to the others. Z3str2 with linear search timed out after 200 s on 31 test cases, whereas CVC4 timed out on 60 test cases. The Z3str2 solver detects overlapping arrangements in both search modes and reports “UNKNOWN” in several cases. However, neither of these solvers crashed or reported errors in any of the test instances.

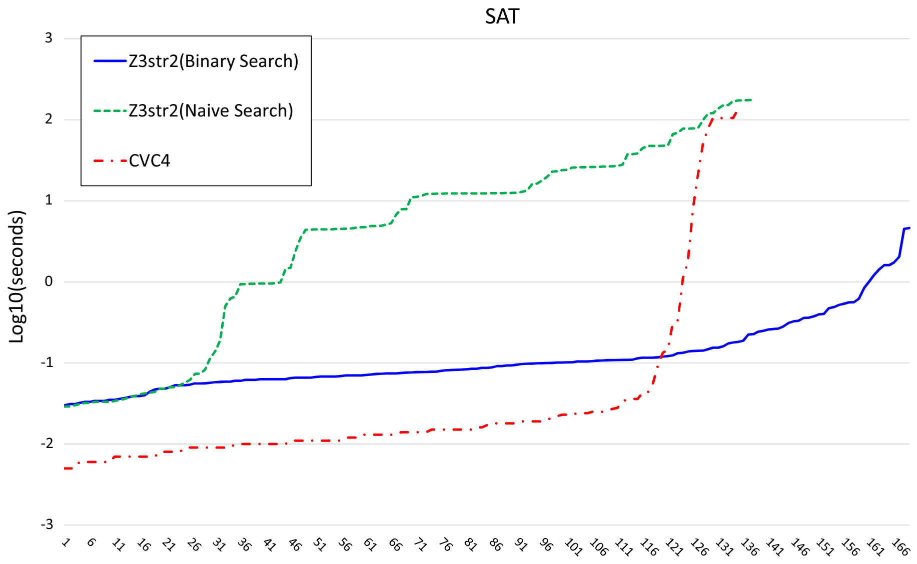


Fig. 19 Cactus plots: SAT instances

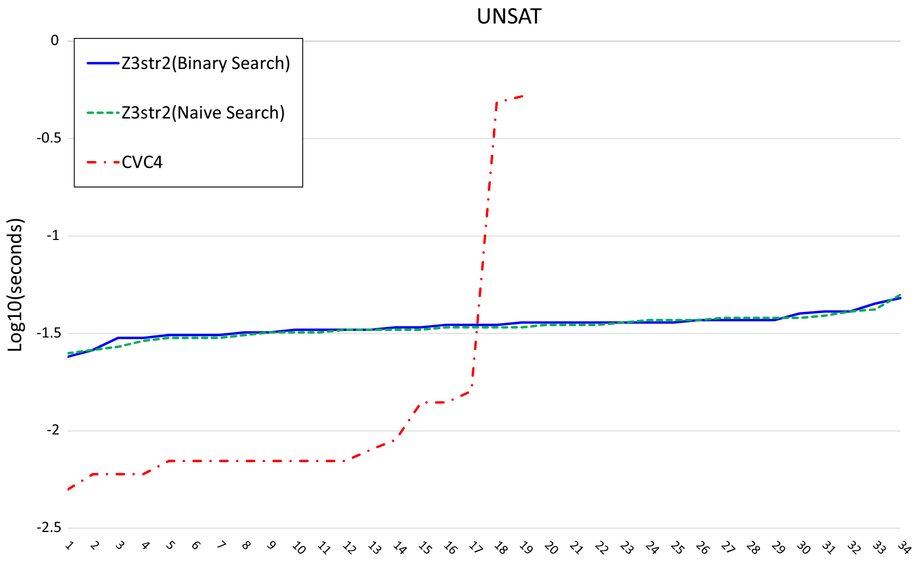


Fig. 20 Cactus plots: UNSAT instances

The cactus plots for the sat and unsat results are presented in the Figs. 19 and 20 respectively. Comparing the time taken by each solver, Z3str2 with the binary search heuristic exhibits the best overall performance. The overall time of the binary search based approach is 41.7 s for these 205 test cases, whereas the linear search method used a total of 9569.6 s and CVC4 needed 12,014.9 s. The binary search based approach used by the new version of Z3str2 is about 229 times faster than the linear search approach of Z3str2 and 288 times faster than CVC4.

9.3 Evaluating improvements in the context of symbolic execution

We demonstrate the performance improvements enabled by a string constraint solver for the combined string-integer theory with respect to the example shown in Fig. 15 by applying both our technique and KLEE, a state-of-the-art symbolic execution engine, to this code. The goal was to detect the heap corruption threat in that code. We faithfully encoded the program snippet in check_login() as string/bit-vector constraints. We then checked whether the buffer pointed to by _username is susceptible to overflow.

Notice that len is an unsigned short variable, and thus ranges from 0 to $2^{16}-1(65{,}535)$. As it represents the buffer size, it determines the number of concrete execution paths KLEE has to enumerate, as well as the search space for library-aware solving. By contrast, the constraints generated by a library-aware SMT solver declaratively model `strlen` as part of the SMT solver logic.

To characterize performance trends, we consider two different precision settings for string length: 8-bit and 16-bit. We used 120 m as the timeout value. There was no need to go beyond 16 bits since KLEE was already significantly slower at 16 bits relative to the library-aware SMT solver. Note that KLEE performs poorly on this test because it has to explore a large number of paths, not because the individual path constraints are difficult to solve (as illustrated in Fig. 21).

The results are provided in Table 8. Under both precision settings, KLEE is consistently and significantly slower than the library-aware solving technique. In particular, if we represent numeric values using 16 bits, then KLEE is not able to find a path that triggers the bug in 120 min, while Z3str2 with binary search heuristics can find a satisfying solution in 787.6 s.

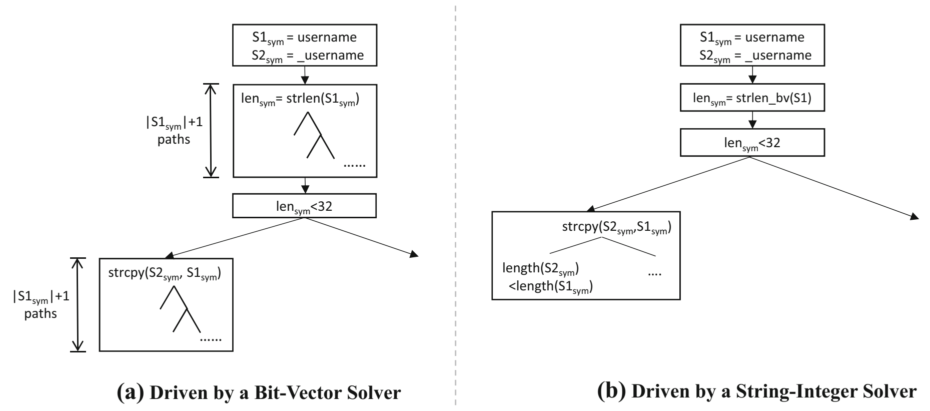


Fig. 21 Comparison between symbolic execution driven by a bit-vector solver (KLEE) and a string solver over the string and integer theories

Table 8 Vulnerability detection using KLEE (solving bit-vector constraints) and Z3str2

Precision	Upper bound	String-integer theory		KLEE (bit-vector)
		Z3str2 (binary search)	Z3str2 (linear search)	
8-bit	255	1.1 s	167 s	300 s
16-bit	65,535	787.6 s	Timeout (7200 s)	Timeout (7200 s)

The benefit thanks to native modeling of strings is clear. Suppose both username and _username are symbolic string variables. In Fig. 21a, as KLEE forks a new state for each character, an invocation of `strlen` on a symbolic string $S1_{sym}$ of size $|S1_{sym}|$ will generate and check $|S1_{sym}| + 1$ path constraints (one per each possible length value between 0 and $|S1_{sym}|$). In Fig. 21b, in contrast, the constraint encoding essentially captures the semantics of the program without explicitly handling the loops in `strlen` and strcmp. Thus, only one query is needed to check whether the length $S2_{sym}$ can be smaller than the length $S1_{sym}$.

10 Related work

The theory considered in this paper—namely the quantifier-free (QF) theory T_{wlr} over word equations, membership predicate over REs, and length function—is a multi-sorted theory with string (str) and numeric (num) sorts. Makanin was the first to show, in 1977, that the QF theory of word equations is decidable [27]. Since, many have improved upon this seminal result [19, 20, 29, 30, 34]. In particular, Plandowski proved that this problem is in PSPACE [30]. Despite decades of effort, the satisfiability problem for T_{wlr} remains open [14, 19, 27, 30]. Still, many practical solvers have been proposed.

Automata-based solvers Regular languages (or automata), as well as context-free grammars (CFGs), can be used to represent strings and handling regex-related operations. JSA [11] computes CFGs for string variables in Java programs. Hooimeijer et al. [16] suggest an optimization whereby automata are built lazily. A primary challenge faced by automata-based approaches, which we do not suffer from, is to capture the connections between strings and other domains, e.g., integers. To overcome this limitation, refinements have been proposed. JST [15] extends JSA. It asserts length constraints in each automaton, and handles numeric constraints after conversion. PISA [35] encodes Java programs into M2L formulas that it discharges to the MONA solver to obtain path- and index-sensitive string approximations. PASS [24, 25] combines automata and parametrized arrays for efficient treatment of unsat cases. Stranger is a powerful extension of string automata with arithmetic automata [38, 39].

Bit-vector based solvers Another group of solvers converts string constraints to constraints into other domains such as integers or bit-vectors. HAMPI [23] is an efficient solver that represents strings as bit-vectors, though it requires the user to provide an upper bound on string lengths. Early versions of Kaluza [33] extended both STP [13] and HAMPI to support mixed string and numeric constraints represented as bit-vector. A similar approach powers Pex [8], though strings are reduced to integer abstractions.

Word-based string solvers CVC4 [26] handles constraints over the theory of unbounded strings with length and RE membership. Solving is based on multi-theory reasoning backed by the DPLL(T) architecture combined with existing SMT theories. The Kleene star operator in RE formulas is dealt with via unrolling as in Z3str2. S3 [36] is another word-based solver, and it can be viewed as an extension of an early version of Z3-str. Roughly speaking, CVC4, S3 and Z3str2 embody similar approaches, and hence CVC4 and S3 can also benefit from the techniques proposed in this paper.

11 Conclusion and future work

We have described three techniques that dramatically improve the efficiency of word-based string solvers: (1) a sound and complete procedure to detect overlapping variables, thereby automatically identifying and avoiding sources of nontermination; (2) tight bi-directional integer/string theory integration, which prunes a vast array of inconsistent search candidates; and (3) further optimization of the string length search strategy based on a binary search heuristic.

We have implemented these techniques on top of Z3-str as Z3str2. We show the efficacy of these techniques through an extensive set of experiments, comparing Z3str2 with the CVC4, S3, Kaluza, PISA and Stranger solvers over four benchmark suites derived from real-world applications. In addition, we empirically show the performance improvements enabled by Z3str2 in the context of symbolic execution of string manipulation programs. We compare the performance of symbolic execution powered by a bit-vector solver and Z3str2. We also evaluate our ability to capture the semantics of string library functions without having to explore a huge number of program execution paths in order to enumerate all possible string lengths.

The technique we present for sound and complete detection of overlapping variables is applicable to other word-based string solvers in addition to Z3str2. As we discussed, the problem of overlapping variables is a key difficulty in reasoning about strings and a source of non-termination in existing tools. The solution we present can be used to avoid these potentially infinite search paths and allow the solve to continue; in future we plan to explore additional heuristics that give the solver the ability to reason about a fragment of cases with overlapping variables in an efficient manner.

Bi-directional integration between the string and integer theory solvers is an extension beyond the classical DPLL(T) model for SMT solvers, which limits the ability for independent theory solvers to communicate. Our experiments have shown that this technique is highly effective at pruning the search space for the solver and preliminary evidence suggests that the binary search heuristic enabled by this integration can improve performance further when dealing with very long strings. We plan to leverage this technique further, and investigate more ways to share information with other theory solvers and additional methods by which our theory solver can guide the search.

While summary-based symbolic execution has been studied previously (e.g. as part of the S-Looper tool [37]), we are not aware of existing work in which SMT solvers are extended with first-class support for programming-language library functions declaratively as part of their logic. One recent application of a similar concept is discussed in [18], in which models of design patterns are abstracted into a symbolic execution engine. The ability to perform a similar analysis at the level of detail of individual library methods can enhance library-aware symbolic execution as part of a library-aware SMT solver. We intend to explore this idea further in the future to broaden its applicability beyond the current context.

References

1. Z3str2 String Constraint Solver. https://sites.google.com/site/z3strsolver/
2. IBM Security AppScan Source. http://www-03.ibm.com/software/products/en/appscan-source
3. Kausler Suite. https://github.com/BoiseState/string-constraint-solvers
4. LibStranger. https://github.com/vlab-cs-ucsb/LibStranger
5. Personal communications with the Stranger team. 2015
6. Abdulla PA, Atig MF, Chen Y-F, Holík L, Rezine A, Rümmer P, Stenman J (2014) String constraints for verification. In: Proceedings of the 26th international conference on computer aided verification, CAV'14, pp 150–166
7. Barrett C, Fontaine P, Tinelli C (2016) The Satisfiability Modulo Theories Library (SMT-LIB). http://www.SMT-LIB.org

8. Bjørner N, Tillmann N, Voronkov A (2009) Path feasibility analysis for string-manipulating programs. In: Proceedings of the 15th international conference on tools and algorithms for the construction and analysis of systems, TACAS '09, pp 307–321
9. Cristian C, Daniel D, Dawson E (2008) Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX conference on operating systems design and implementation, OSDI'08. USENIX Association, Berkeley, pp 209–224
10. Chipounov V, Kuznetsov V, Candea G (2011) S2e: a platform for in-vivo multi-path analysis of software systems. In: Proceedings of the sixteenth international conference on architectural support for programming languages and operating systems. ASPLOS XVI. ACM, New York, pp 265–278
11. Christensen AS, Møller A, Schwartzbach MI (2003) Precise analysis of string expressions. In: Proceedings of the 10th international conference on static analysis, SAS'03, pp 1–18
12. De Moura L, Bjørner N (2008) Z3: an efficient smt solver. In: Proceedings of the theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems, TACAS'08, pp 337–340
13. Ganesh V, Dill DL (2007) A decision procedure for bit-vectors and arrays. In: Proceedings of the 19th international conference on computer aided verification, CAV'07, pp 519–531
14. Ganesh V, Minnes M, Solar-Lezama A, Rinard M (2012) Word equations with length constraints: what's decidable? In: HVC'12
15. Ghosh I, Shafiei N, Li G, Chiang W-F (2013) JST: an automatic test generation tool for industrial java applications with strings. In: Proceedings of the 2013 international conference on software engineering, ICSE '13, pp 992–1001
16. Hooimeijer P, Weimer W (2010) Solving string constraints lazily. In: Proceedings of the IEEE/ACM international conference on automated software engineering, ASE '10, pp 377–386
17. Hopcroft JE, Motwani R, Ullman JD (2007) Introduction to automata theory, languages, and computation. Pearson/Addison Wesley, Reading
18. Jeon J, Qiu X, Fetter-Degges J, Foster JS, Solar-Lezama A (2016) Synthesizing framework models for symbolic execution. In: Proceedings of the 38th international conference on software engineering, ICSE '16. ACM, New York, pp 156–167
19. Jeż A (2013) Recompression: word equations and beyond. In: Developments in language theory, Lecture Notes in Computer Science, pp 12–26
20. Karhumäki J, Mignosi F, Plandowski W (2000) The expressibility of languages and relations by word equations. J ACM 47(3):483–505
21. Kausler S (2014) Evaluation of string constraint solvers using dynamic symbolic execution. Master's Thesis, Boise State University
22. Kausler S, Sherman E (2014) Evaluation of string constraint solvers in the context of symbolic execution. In: Proceedings of the 29th ACM/IEEE international conference on automated software engineering, ASE '14. ACM, New York, pp 259–270
23. Kiezun A, Ganesh V, Guo PJ, Hooimeijer P, Ernst MD (2009) Hampi: a solver for string constraints. In: Proceedings of the eighteenth international symposium on software testing and analysis, ISSTA '09, pp 105–116
24. Li G, Andreasen E, Ghosh I (2014) SymJS: automatic symbolic testing of javascript web applications. In: Proceedings of the 22Nd ACM SIGSOFT international symposium on foundations of software engineering, FSE 2014, pp 449–459
25. Li G, Ghosh I (2013) PASS: string solving with parameterized array and interval automaton. In: 9th international Haifa verification conference, HVC '13, pp 15–31
26. Liang T, Reynolds A, Tinelli C, Barrett C, Deters M (2014) A dpll(t) theory solver for a theory of strings and regular expressions. In: Proceedings of the 26th international conference on computer aided verification, CAV'14, pp 646–662
27. Makanin GS (1977) The problem of solvability of equations in a free semigroup. Math Sbornik 103:147–236 (1977). English transl. in Math USSR Sbornik 32
28. Matiyasevich Y (2007) Word equations, Fibonacci numbers, and Hilbert's tenth problem. In: Workshop on Fibonacci words
29. Plandowski W (2004) Satisfiability of word equations with constants is in pspace. J ACM 51(3):483–496
30. Plandowski W (2006) An efficient algorithm for solving word equations. In: Proceedings of the 38th annual ACM symposium on theory of computing, STOC '06, pp 467–476
31. Qu X, Robinson B (2011) A case study of concolic testing tools and their limitations. In: 2011 international symposium on empirical software engineering and measurement (ESEM). IEEE Computer Society, pp 117–126

32. Redelinghuys G, Visser W, Geldenhuys J (2012) Symbolic execution of programs with strings. In: Proceedings of the South African Institute for computer scientists and information technologists conference, SAICSIT '12, pp 139–148
33. Saxena P, Akhawe D, Hanna S, Mao F, McCamant S, Song D (2010) A symbolic execution framework for javascript. In: Proceedings of the 2010 IEEE symposium on security and privacy, SP '10, pp 513–528
34. Schulz K (1992) Makanin's algorithm for word equations-two improvements and a generalization. In: Schulz K (ed) Word equations and related topics, Lecture Notes in Computer Science, vol 572. Springer, Berlin, pp 85–150
35. Tateishi T, Pistoia M, Tripp O (2013) Path- and index-sensitive string analysis based on monadic second-order logic. ACM Trans Softw Eng Methodol 22(4):33:1–33:33
36. Trinh M-T, Chu D-H, Jaffar J (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, pp 1232–1243
37. Xie X, Liu Y, Le W, Li X, Chen H (2015) S-looper: automatic summarization for multipath string loops. In: Proceedings of the 2015 international symposium on software testing and analysis, ISSTA 2015. ACM, New York, pp 188–198
38. Yu F, Alkhalaf M, Bultan T (2010) Stranger: an automata-based string analysis tool for php. In: Proceedings of the 16th international conference on tools and algorithms for the construction and analysis of systems, TACAS'10, pp 154–157
39. Yu F, Bultan T, Ibarra OH (2009) Symbolic string verification: combining string analysis and size analysis. In: Proceedings of the 15th international conference on tools and algorithms for the construction and analysis of systems, TACAS '09, pp 322–336
40. Zheng Y, Ganesh V, Subramanian S, Tripp O, Dolby J, Zhang X (2015) Effective search-space pruning for solvers of string equations, regular expressions and length constraints. Technical report. https://sites.google.com/site/z3strsolver/publications
41. Zheng Y, Zhang X, Ganesh V (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, pp 114–124