Path Feasibility Analysis for String-Manipulating Programs

Nikolaj Bjørner[1], Nikolai Tillmann[1] and Andrei Voronkov[2]

[1] Microsoft Research http://research.microsoft.com
[2] University of Manchester http://www.voronkov.com

Abstract. We discuss the problem of path feasibility for programs manipulating strings using a collection of standard string library functions. We prove results on the complexity of this problem, including its undecidability in the general case and decidability of some special cases. In the context of test-case generation, we are interested in an efficient finite model finding method for string constraints. To this end we develop a two-tier finite model finding procedure. First, an integer abstraction of string constraints are passed to an SMT (Satisfiability Modulo Theories) solver. The abstraction is either unsatisfiable, or the solver produces a model that fixes lengths of enough strings to reduce the entire problem to be finite domain. The resulting fixed-length string constraints are then solved in a second phase. We implemented the procedure in a symbolic execution framework, report on the encouraging results and discuss directions for improving the method further.

1 Introduction

Dynamic symbolic execution [8, 4, 15, 17] has recently gained attention in the context of test-case generation. It extends static symbolic execution [12] by collecting symbolic constraints from concrete execution traces obtained by monitoring the executed instructions. In order to explore a different execution path it suffices to modify one of the extracted symbolic traces by selecting and negating a branch condition, which we call *flipping a branch*. Then a constraint solver is used to provide a satisfying assignment to the modified path condition.

Strings form a fundamental data type found in most if not all general purpose programming languages. Strings may be represented in various ways, such as a pointer to a 0-terminated array of characters (in C), as an array object with an explicit length (in Java and C#), or even as a singly linked list (in Haskell). Programs that manipulate strings can often abstract from the representation and use a set of library routines to perform common functions on strings, such as converting characters to strings, finding characters and extracting substrings.

The problem String library routines are themselves implemented as programs, so dynamic symbolic execution can apply to string routines by exploring the underlying programs and solving constraints on the data types used in these programs. There is an inherent overhead of this approach, as the general dynamic symbolic exploration engine has to search the state space of the string library routines for solutions to string constraints. We can take advantage of the fact that path constraints from common string library routines have a mathematical abstraction rooted in word equations. This suggests that we may work at the level of abstract strings and treat calls to the string library functions as operations in a *theory* of strings. As we will see, the full set of constraints that can be created from common string functions do not fall in a decidable class. On the other hand, our objective is really to find small strings that can be supplied as unit tests, so an incremental small and finite model-finding routine provides the right match.

The main existing way of handling strings in symbolic test-case generation tools is to *not* handle them specially. For example in the current release of Pex [17], strings are represented as arrays and string library routines are explored like any other procedure. As a result, conceptually simple calls to library functions become programs containing loops. Summaries [7] provide one additional layer on top of the string procedures to allow the search on feasible paths to coalesce several traversals of the same procedure body.

Example Consider the program shown in Figure 1. This program checks whether the input string s is a URL encoding a query about EasyChair to either Microsoft Live Search or Google. Essentially, such a string must start with "http://" followed by a domain of one of the search engines, followed by a "/", and after this "/" it should contain a substring "EasyChair" and not contain other "/".

```
private bool IsEasyChairQuery(string str)
{
  // (1) check that str contains "/" followed by anything not
  // containing "/" and containing "EasyChair"
  int lastSlash = str.LastIndexOf('/');
  if (lastSlash < 0){return false;}
  var rest = str.Substring(lastSlash + 1);
  if (! rest.Contains("EasyChair")){return false;}
  // (2) Check that str starts with "http://"
  if (! str.StartsWith("http://")){return false;}
  // (3) Take the string between "http://" and the last "/".
  // if it starts with "www." strip the "www." off
  var t = str.Substring("http://".Length,
                        lastSlash - "http://".Length);
  if (t.StartsWith("www.")){t = t.Substring("www.".Length);}
  // (4) Check that after stripping we have either "live.com"
  // or "google.com"
  if (t != "live.com" && t != "google.com"){return false; }
  // s survived all checks
  return true;
}
```

Fig. 1. The EasyChair Query program

We will use this program as the running example.

Example 1. Consider the following path in the program, where queries are denoted by "?".

```
lastSlash = str.LastIndexOf('/');
? ¬ lastSlash < 0
rest = str.Substring(lastSlash + 1);
? rest.Contains("EasyChair")
? str.StartsWith("http://")
t = str.Substring("http://".Length,lastSlash - "http://".Length);
? t.StartsWith("www.")
t = t.Substring("www.".Length);
? t = "live.com"
```

We are interested in checking feasibility of this path, that is, finding an input string `str` so that the program run with this string as an input will follow this path.

The rest of this paper is organized as follows. Above we introduced a running example illustrating path constraints from a simple string-manipulating program. Section 2 defines path feasibility in the context of constraints from basic .NET string library functions and defines a first-order string library language corresponding to these library functions. The resulting constraints for all but one library function are compiled into the core language as outlined in Section 3. Section 4 discusses the decidability of the library language. One fragment is equivalent to a long standing open problem in word equations, another fragment is shown undecidable. This confirms the complexity of the path feasibility problem for string library functions. We also show that a so-called *fixed-length* fragment is decidable: this fact is used in our implementation. We point out that two other decidable fragments can be obtained using results from word equations and automatic structures. Section 5 outlines our incremental procedure for enumerating solutions to core language constraints. Sections 6 and 7 describe an implementation and its evaluation.

2 Path Feasibility and String Constraints

In this section we define a language for representing the path feasibility problem as a constraint satisfaction problem.

The String Library Language In this section we will introduce a *string library language* $\mathcal{LL}$. This language is a first-order language for describing constraints involving string library functions. The string library we are interested in is the `.NET` String library, however, we believe that other string libraries can be captured by similar languages and processed using the methods described in this paper.

Let $\mathbb{C}$ be a finite set of *characters* and $\mathbb{S}$ the set of all strings built using these characters. By $\mathbb{I}$ we denote the set of all integers. The *string library language* $\mathcal{LL}$ is a many-sorted first-order language defined as follows. The language has the following three sorts: the *character sort* $\mathcal{C}$, the *string sort* $\mathcal{S}$, the *integer sort* $\mathcal{I}$ and the Boolean sort $\mathcal{B}$. It also contains a countably infinite number of variables of each sort.

function	type	meaning
Chars(s, i)	$\mathcal{S} \times \mathcal{I} \to \mathcal{C}$	the character at position i in s
Compare(s_1, s_2)	$\mathcal{S} \times \mathcal{S} \to \mathcal{I}$	string comparison
Concat(s_1, s_2)	$\mathcal{S} \times \mathcal{S} \to \mathcal{S}$	string concatenation
Contains(s_1, s_2)	$\mathcal{S} \times \mathcal{S} \to \mathcal{B}$	true if s_2 is a substring of s_1
Equals(s_1, s_2)	$\mathcal{S} \times \mathcal{S} \to \mathcal{B}$	true if $s_1 = s_2$
IndexOf$_4(s_1, s_2, i)$	$\mathcal{S} \times \mathcal{S} \times \mathcal{I} \to \mathcal{I}$	the index of the first occurrence of s_2 in s_1 starting at position i or -1 if not found
LastIndexOf$_1(s, c)$	$\mathcal{S} \times \mathcal{C} \to \mathcal{I}$	the index of the last occurrence of c in s or -1 if not found
Length(s)	$\mathcal{S} \to \mathcal{I}$	the length of s
Replace$_2(s_1, s_2, s_3)$	$\mathcal{S} \times \mathcal{S} \times \mathcal{S} \to \mathcal{S}$	replace all occurrences of s_2 in s_1 with s_3
StartsWith(s_1, s_2)	$\mathcal{S} \times \mathcal{S} \to \mathcal{B}$	true if s_1 starts with s_2
Substring$_1(s, i)$	$\mathcal{S} \times \mathcal{I} \to \mathcal{S}$	substring of s starting at position i
Substring$_2(s, i_1, i_2)$	$\mathcal{S} \times \mathcal{I} \times \mathcal{I} \to \mathcal{S}$	substring of s starting at position i_1 and having length i_2
ToString(c)	$\mathcal{C} \to \mathcal{S}$	string consisting of a single character c
ToUpper(s)	$\mathcal{S} \to \mathcal{S}$	string obtained by converting s to upper case

Table 1. String library functions

The language $\mathcal{LL}$ contains constants denoting all integers, characters and strings, and some functions on strings and integers. Table 1 contains a description of a representative subset of the `.NET` string library. In this description x_i denotes the ith argument of a function. Positions in strings are number from 0 so the character at the position 0 in a string is always the first character of the string.

Some functions of the `.NET` library are overloaded. To resolve ambiguity, we add indexes to the names of these functions. For example, there are six different `.NET` functions having the name *IndexOf*. In $\mathcal{LL}$ we add indexes to function names to distinguish them (cf. *IndexOf*$_4$ in the table).

To define the semantics of $\mathcal{LL}$ we introduce a notion of value assignment. A *value assignment* for this language is a mapping of variables to $\mathbb{C} \cup \mathbb{S} \cup \mathbb{I}$ so that every variable is mapped to an element of the corresponding domain, for example, variables of the sort $\mathcal{I}$ are mapped to integer values. We will now extend value assignments to arbitrary expressions and quantifier-free formulas. To this end we should take care of undefined values since some of the library functions are partial and calling them outside of their domain causes an exception. We will specify the exception conditions later.

Let us introduce a special *undefined value* $\perp$ and extend value assignments to arbitrary expressions of $\mathcal{LL}$ as follows. For every value assignment v and expression e different from a variable we define $v(e)$ as follows.

1. If e is a constant, then $v(e)$ is the value of this constant, for example the value of the integer constant 7 is the number 7.
2. Suppose that e has the form $f(e_1, \ldots, e_n)$ where f is a function of $\mathcal{LL}$ and let $v_1 = v(e_1), \ldots, v_n = v(e_n)$. If for some $i \in \{1, \ldots, n\}$ we have $v_i = \perp$, then $v(e) = \perp$. Otherwise, let the $\widehat{f}$ be the function corresponding to f (see Table 1 for the definition of these functions). If $\widehat{f}$ is undefined on $(v_1, \ldots, v_n)$, then $v(e) = \perp$, otherwise $v(e)$ if the value $\widehat{f}(v_1, \ldots, v_n)$.

If $v(e) = \bot$, we say that $v(e)$ is *undefined* under the value assignment v.

The next step is to define the semantics of formulas. To this end we will use the three-valued logic having the Boolean values `true` and `false` and the undefined value $\bot$. In this logic, for example, a disjunction is true if at least one of its members is true; false if all of them are false and undefined otherwise. We leave out details due to a lack of space.

Path Feasibility Suppose that π is a path in a program using only assignments and tests. The *path feasibility problem* is the problem of finding initial values of variables which makes the path feasible, that is the values so that for the program execution starting with this assignment satisfies all tests on the paths and raises no exceptions. The path feasibility problem can be reduced to the following *constraint satisfaction problem for* $\mathcal{LL}$:

> Given a finite set $\{L_1, \ldots, L_n\}$ of literals, find a value assignment v which makes all of these literals true.

Evidently, the path feasibility problem can be reduced in a straightforward way to the constraint satisfaction problem for $\mathcal{LL}$. We do not give full details here but restrict ourselves to an example.

Example 2. Consider the path of Example 1. This path can be translated to the following constraint.

$$\begin{array}{ll}
i_1 = \mathit{LastIndexOf}_1(s_1, \texttt{'/'}) & i_2 = \mathit{Length}(\texttt{"http://"}). \\
\neg i_1 < 0. & s_3 = \mathit{Substring}_2(s_1, \ell, i_1 - i_2). \\
s_2 = \mathit{Substring}_1(s_1, i_1 + 1). & \mathit{StartsWith}(s_3, \texttt{"www."}). \\
\mathit{Contains}(s_2, \texttt{"EasyChair"}). & s_4 = \mathit{Substring}_1(s_3, \mathit{Length}(\texttt{"www."})). \\
\mathit{StartsWith}(s_1, \texttt{"http://"}). & s_4 = \texttt{"live.com"}.
\end{array}$$

This constraint is satisfiable if and only if there exists an initial value of the variable s_1 so that the program will follow the intended path.

3 The Core String Language

In this section we will define the so-called *core string language* $\mathcal{CL}$. This language contains a smaller number of functions than $\mathcal{LL}$ but we will be interested in a larger fragment of this language, including propositional connectives and *bounded quantifiers* of a special form. This language will play the role of an intermediate language between the string library language and an SMT solver. We will start with defining $\mathcal{CL}$ and giving a translation of $\mathcal{LL}$ into $\mathcal{CL}$. This translation will also define the semantics of the $\mathcal{LL}$ functions in a strict way as opposed to a less formal description in Table 1.

Definition 1 (core string language). The core string language contains the following functions:

1. The string functions *Length* and *Chars*. We will write $\ell(s)$ instead of *Length*(s) and $s[i]$ instead of *Chars*(s, i).
2. The standard functions and predicates of linear integer arithmetic.
3. Some functions and predicates on characters. We do not specify the set of all these functions but will use two of them (comparison $<$ and *ToUpper*) later.

The *formulas* of the language are defined as follows:

1. Every atomic formula is a formula;
2. If F_1 and F_2 are formulas, then $\neg F_1$, $F_1 \wedge F_2$, $F_1 \vee F_2$ and $F_1 \rightarrow F_2$ are formulas.
3. If F is a formula, i a variable of the sort $\mathcal{I}$ and e_1, e_2 expressions of the sort $\mathcal{I}$ not containing i, then $(\forall i)(e_1 \leq i \wedge i \leq e_2 \rightarrow F)$ and $(\exists i)(e_1 \leq i \wedge i \leq e_2 \wedge F)$ are formulas.

We will write $(\forall i \in [e_1 \dots e_2])F$ and $(\exists i \in [e_1 \dots e_2])F$ respectively instead of $(\forall i)(e_1 \leq i \wedge i \leq e_2 \rightarrow F)$ and $(\exists i)(e_1 \leq i \wedge i \leq e_2 \wedge F)$. Note the following equivalences:

$$\neg(\forall i \in [e_1 \dots e_2])F \equiv (\exists i \in [e_1 \dots e_2])\neg F;$$
$$\neg(\exists i \in [e_1 \dots e_2])F \equiv (\forall i \in [e_1 \dots e_2])\neg F.$$

For simplicity we will usually say *the library language* instead of the string library language and *the core language* instead of the core string language. Likewise, we will simply say *library functions* and *core functions*.

Function	exception condition
Chars(s, i)	$i < 0 \vee i \geq \ell(s)$
IndexOf$_4(s_1, s_2, i)$	$i < 0 \vee \ell(s_1) \geq i$
Replace$_2(s_1, s_2, s_3)$	$\ell(s_2) = 0$
Substring$_1(s, i)$	$i < 0 \vee i > \ell(s)$
Substring$_2(s, i_1, i_2)$	$i_1 < 0 \vee i_2 < 0 \vee i_1 + i_2 > \ell(s)$

Table 2. Exception conditions for the library functions

Let us now define formally the translation of the library language into the core language. We will do this by first defining exception conditions for all library functions and then the library functions themselves using the core language. The exception conditions are given in Table 2. Note that ℓ is the only non-arithmetical core function used in the exception conditions.

Let us introduce two abbreviations for the core language as follows.

$$s_1[i \dots j] = s_2 \overset{\text{def}}{=} \ell(s_2) = j - i + 1 \wedge (\forall k \in [i \dots j])(s_1[k] = s_2[k - i]). \quad (1)$$
$$s_1 \sqsubseteq s_2 \overset{\text{def}}{=} (\exists i \in [0 \dots \ell(s_2) - \ell(s_1)])s_2[i \dots i + \ell(s_1) - 1] = s_1. \quad (2)$$

One can easily show that $s_1 \sqsubseteq s_2$ is equivalent to *Contains*(s_2, s_1) and to $\exists s_3 \exists s_4(s_3 \cdot s_1 \cdot s_4 = s_2)$. The difference is that $s_1 \sqsubseteq s_2$ is a formula of the core language.

Let us now give a definition for every library function in the core language. This, together with the definition of exception conditions, also provides a precise semantics

function	definition
$\mathit{Chars}(s, i)$	$s[i]$
$\mathit{Compare}(s_1, s_2) = 0$	$s_1 = s_2$
$\mathit{Compare}(s_1, s_2) < 0$	$(\exists i \in [0 \dots \ell(s_1) - 1])(\,\ell(s_2) > i \land s_1[0 \dots i-1] = s_2[0 \dots i-1] \land$ $(\ell(s_1) = i \lor s_1[i] < s_2[i]))$
$\mathit{Compare}(s_1, s_2) > 0$	$(\exists i \in [0 \dots \ell(s_2) - 1])(\,\ell(s_1) > i \land s_1[0 \dots i-1] = s_2[0 \dots i-1] \land$ $(\ell(s_2) = i \lor s_2[i] < s_1[i]))$
$\mathit{Concat}(s_1, s_2)$	$s_1 \cdot s_2$
$\mathit{Contains}(s_1, s_2)$	$s_2 \sqsubseteq s_1$
$\mathit{Equals}(s_1, s_2)$	$s_1 = s_2$
$\mathit{IndexOf}_4(s_1, s_2, i_1) = i_0$	$(i_0 = -1 \land s_2 \not\sqsubseteq s_1[i_1 \dots \ell(s_1) - 1]) \lor$ $(i_0 \geq i_1 \land s_1[i_0 \dots i_0 + \ell(s_2) - 1] = s_2 \land$ $(\forall j \in [i_1 \dots i_0 - 1])(s_1[j \dots j + \ell(s_2) - 1] \neq s_2))$
$\mathit{LastIndexOf}_1(s, c) = i$	$(i = -1 \land (\forall j \in [0 \dots \ell(s) - 1])s[j] \neq c) \lor$ $(i \geq 0 \land s[i] = c \land (\forall j \in [i+1 \dots \ell(s) - 1])s[j] \neq c)$
$\mathit{Replace}_2(s_1, s_2, s_3)$	no definition exists
$\mathit{StartsWith}(s_1, s_2)$	$(\exists i \in [0 \dots \ell(s_1) - \ell(s_2)])s[0 \dots i-1] = s_2$
$\mathit{Substring}_1(s, i)$	$s[i \dots \ell(s) - 1]$
$\mathit{Substring}_2(s, i_1, i_2)$	$s[i_1 \dots i_1 + i_2 - 1]$
$\mathit{ToString}(c) = s$	$\ell(s) = 1 \land s[0] = c$
$\mathit{ToUpper}(s_1) = s_2$	$\ell(s_2) = \ell(s_1) \land (\forall i \in [0 \dots \ell(s_1) - 1])\mathit{Upper}(s_1[i], s_2[i])$

Table 3. Definitions of some library predicates

for the library functions instead of the less formal explanation given in Table 1. The definitions of the library functions are given in Table 3.

We are interested in the following fragment of the core language.

Definition 2. Let F be a formula of either the library or the core language such that (i) the string variables occurring in F are $s_1, \dots, s_n$ and (ii) F contains no free occurrences of integer variables. The formula F is said to be a *fixed-length formula* if it has the form

$$\ell(s_1) = i_1 \land \dots \land \ell(s_n) = i_n \land F',$$

where $i_1, \dots, i_n$ are concrete integer constants. The *fixed-length fragment* of the library (respectively, core) language is the set of all fixed-length formulas of this language.

Theorem 1. *The satisfiability problem for the fixed-length fragment of the core language is decidable.*

Proof. Since $i_1, \dots, i_n$ are integer constants, every value assignment that satisfies the formula assigns to s_k a string of the length i_k, for all $k = 1 \dots n$. There exists only a finite number of value assignments with this property, so by substituting these value assignments the formula can be replaced by a finite disjunction of formulas with no free variables. It remains to show that satisfiability of closed formulas of the core language is decidable. This can be proved by induction on the depth of quantifiers in a formula F. If the number of quantifiers in F is 0, F is a variable-free quantifier-free formula which can simply be evaluated. Suppose now that F has at least one quantifier, we will then show how to eliminate a quantifier in F. Take any quantified subformula G of F that is not in the scope of another quantifier. Without loss of generality we can assume that G is of the form $(\forall i \in [e_1 \ldots e_2])H(i)$. Then e_1 and e_2 are variable free, so they can be evaluated to concrete integer values k_1 and k_2, hence G can be replaced by the conjunction $\bigwedge_{k_1 \leq k \leq k_2} H(k)$ having a smaller quantifier depth.

Note that this proof works for the string library with any finite number of functions or predicates on characters, since for such functions we only need that they can be computed. There is another, more efficient decision procedure for this fragment used in our implementation.

As a consequence of this theorem we obtain the following result.

Theorem 2. *The satisfiability problem for the fixed-length fragment of the library language without the function Replace is decidable.*

Proof. Using definitions of Table 3 one can translate any formula of the library language without the function *Replace* into an equivalent formula of the core language. Since the translation does not introduce new free variables, any fixed-length formula is translated into a fixed-length formula. Then apply Theorem 1 on the decidability of the fixed-length fragment of the core language.

Theorem 2 and its proof provide a foundation for our method of constraint solving.

4 Decidability and Undecidability Results for the Library Language

In this section we consider the full (not fixed-length) library language. We will show that constraint solving for this language is undecidable. We will also point out that the decidability of a very large fragment of this language is equivalent to the decidability problem of a well-known problem related to word equations. Finally, we will prove some decidability results.

It is easy to see that several functions and predicates of the string library can be expressed using string concatenation. Among the representative subset selected by us these are *Concat*, *Contains*, *Equals* and *StartsWith*. Solving constraints using only such functions and predicates can be reduced to solving word equations and so is decidable. However, this fragment is hardly very practical.

Word Equations and Equal Length Constraints Let us call the subset of $\mathcal{LL}$ without *Replace*, *ToUpper* the *pure library language*. It is interesting that path feasibility in the pure library language is equivalent to a well-known extension of word equations whose decidability is an open problem. This problem is known as *word equations with the equal length predicate*. The equal length predicate, denoted by $\ell\ell$ is true on a pair of strings s_1, s_2 if and only if s_1 and s_2 have the same length. The decidability of word equations with the equal length predicate is an open problem [3, 13].

Theorem 3. *The path feasibility problem in the pure library language is decidable if and only if word equations with the equal length predicate are decidable.*

Proof. We will show how to reduce the two problems to each other. To this end, we will first reformulate the constraint satisfaction problem for $\mathcal{LL}$ as a problem on strings. To represent a character c as a string we will use the string consisting of this character. To represent non-negative integers, we will use the following idea. Let us fix a letter of the alphabet, for example, `|`. We will use the string `|` ... `|` of the length n, denoted by $\widehat{n}$ as a representation for the number n and call such strings *numerals*. To represent a negative number $-n$ we can use, for example, the string `-|` ... `|` of the length $n+1$, that is `-`$\widehat{n}$. Let us prove several facts about this representation. When we write that some relation can be represented we mean that it can be represented using an existential formula. Note the well-known fact [3] that inequations of words can be represented using equations, hence we will not consider negative literals in the proof.

Using word equations one can express that a string s is a numeral. Indeed, consider the equation s`|` $=$ `|`s. The solutions of this equation are exactly all numerals.

Using word equations one can express the addition on integers. In our representation the addition on non-negative integers becomes string concatenation: indeed, $\widehat{m}\widehat{n} = \widehat{m+n}$ for all non-negative integers m, n. It is not hard to represent addition on all integers too.

One can express the equal length predicate using the length function and vice versa. One direction is obvious since $\ell\ell(s_1, s_2) \equiv \ell(s_1) = \ell(s_2)$. In the other direction, note that the length of a string s is equal to n if and only if s and $\widehat{n}$ have equal length.

What remains to note is that for the pure library language functions both the equality and the inequality between these functions can be represented using string concatenation and the length function.

Let us now prove an undecidability result.

Theorem 4. *The path feasibility problem for the library language is undecidable.*

Proof. For every character c of the alphabet consider the following function ℓ_c: for every string s, $\ell_c(s) = \widehat{n}$, where n is the number of occurrences of c in n. We will use a following result from [3]: the existential theory of words with concatenation, the equal length predicate and functions ℓ_0, ℓ_1 is undecidable. We already proved in Theorem 3 that the equal length predicate is expressible in the string library language, it remains to note that using ℓ_c for every character c is expressible using the library function *Replace*.

One can prove other decidability results about using automatic structures, we will only briefly sketch how one can prove them here. Let us call a predicate or a function on strings *automatic* if it can be represented by a finite automaton, for details see [11]. To apply this definition to integers and characters we assume that they are represented respectively as numerals and as one-character strings. The string library contains several automatic functions, namely *Chars*, *Compare*, *Equals*, *Length*, *StartsWith*, *ToString*, *ToUpper*. Other functions are not automatic but have automatic instances when one or more arguments are instantiated to constants. For example, for every constant string s_2, *Concat*(s_1, s_2) considered as a function of s_1 is automatic. It is also automatic if we fix the first argument s_1 to be constant. It is known that the full first-order theory of every automatic structure (structure in which all predicates and functions are automatic) is decidable. However, this result is hardly interesting in practice for two reasons. Firstly, using automata-based method on large alphabets is prohibitively expensive. Secondly, the resulting decidable fragment is too narrow for applications, for example, it does not include concatenation and integer addition.

5 Solving constraint satisfaction problems in the target language

Our algorithm for checking constraint satisfaction is described here and works as follows. First, we flatten the input constraint obtaining a flattened constraint C. After that, we produce a so-called *integer abstraction* of the problem. The integer abstraction is a quantifier-free formula I of linear arithmetic over two kinds of integer variable: those coming from the constraint C and those denoting the lengths of string variables occurring in C. After that we look for small solutions to I. If there is no such solution, then the original constraint is unsatisfiable. If I has a solution it gives us the lengths of all string variables in C. When we fix the length, we obtain a fixed-length formula in the core language which can be decided by a finite domain CSP or a SAT solver. If this formula has no solution, backtrack and try to find another solution of I.

Flattening A constraint C is called *flat* if all string library functions occur in C at the top level, that is, for every term of formula $p(t_1, \ldots, t_n)$ occurring in C, where p is different from equality, the terms $t_1, \ldots, t_n$ contain no occurrences of the string library functions. For example, the constraint $\mathit{Substring}_1(s_1, i_1 + i_2)$ is flat while the constraint $s_1 = \mathit{Concat}(s_2, \mathit{Substring}_1(s_3, 1))$ is not, since $\mathit{Substring}_1$ does not occur at the top level. One can change any constraint into a constraint all whose literals are flat by introducing extra variables for subterms. For example, the constraint of Example 2 will become as shown above.

$i_1 = \mathit{LastIndexOf}_1(s_1, \texttt{'/'})$.
$\neg i_1 < 0$.
$s_2 = \mathit{Substring}_1(s_1, i_1 + 1)$.
$\mathit{Contains}(s_2, \texttt{"EasyChair"})$.
$\mathit{StartsWith}(s_1, \texttt{"http://"})$.
$i_2 = \mathit{Length}(\texttt{"http://"})$.
$s_3 = \mathit{Substring}_2(s_1, i_2, i_1 - i_2)$.
$\mathit{StartsWith}(s_3, \texttt{"www."})$.
$i_3 = \mathit{Length}(\texttt{"www."})$.
$s_4 = \mathit{Substring}_1(s_3, i_3)$.
$s_4 = \texttt{"live.com"}$

Integer abstraction The integer abstraction of a literal defines necessary conditions for the literal to be true. This implies that every solution to the literal must also be a solution to the integer abstraction. For every literal L in a flat constraint its integer abstraction is built as follows. First, let F^e be the exception condition corresponding to the literal L in Table 2; F^e is `false` if there are no matching exception conditions for L. If the literal L matches an entry in Table 4 with formula F^i, then the integer abstraction of L is given as $\neg F^e \wedge F^i$. If L is a negation $\neg L'$ and L' matches an entry in Table 4, then the abstraction is given as $\neg F^e$; otherwise, the abstraction of L is set to L.

The flat constraint for our running example has the integer abstraction given in Figure 2.

function	abstraction
$\mathit{Compare}(s_1, s_2) = c$	$(\ell(s_1) = 0 \rightarrow c \leq 0) \wedge (\ell(s_2) = 0 \rightarrow c \geq 0)$
$\mathit{Concat}(s_1, s_2) = s$	$\ell(s) = \ell(s_1) + \ell(s_2)$
$\mathit{Contains}(s_1, s_2)$	$\ell(s_1) \geq \ell(s_2)$
$\mathit{IndexOf}_4(s_1, s_2, i_1) = i$	$i = -1 \vee (i \geq i_1 \wedge i + \ell(s_2) \leq \ell(s_1))$
$\mathit{LastIndexOf}_1(s, c) = i$	$i = -1 \vee i < \ell(s)$
$\mathit{Replace}_2(s_1, s_2, s_3) = s_0$	$(\ell(s_2) \geq \ell(s_3) \rightarrow \ell(s_1) \geq \ell(s_0)) \wedge$ $(\ell(s_2) \leq \ell(s_3) \rightarrow \ell(s_1) \leq \ell(s_0))$
$\mathit{StartsWith}(s_1, s_2)$	$\ell(s_1) \geq \ell(s_2)$
$\mathit{Substring}_1(s_1, i) = s_0$	$\ell(s_0) = \ell(s_1) - i$
$\mathit{Substring}_2(s_1, i, j) = s_0$	$\ell(s_0) = j \wedge \ell(s_1) \geq i + j$
$\mathit{ToString}(c) = s$	$\ell(s) = 1$
$\mathit{ToUpper}(s_1) = s_0$	$\ell(s_0) = \ell(s_1)$

Table 4. Integer abstraction of library predicates

$i_1 = \mathit{LastIndexOf}_1(s_1, \texttt{'/'})$	$i_1 = -1 \vee i_1 < \ell(s_1)$
$\neg i_1 < 0$	$\neg i_1 < 0$
$s_2 = \mathit{Substring}_1(s_1, i_1 + 1)$	$\neg(i_1 + 1 < 0 \vee i_1 + 1 > \ell(s_1)) \wedge \ell(s_2) = \ell(s_1) - i_1 - 1$
$\mathit{Contains}(s_2, \texttt{"EasyChair"})$	$\ell(s_2) \geq 9$
$StartsWith(s_1, \texttt{"http://"})$	$\ell(s_1) \geq 7$
$i_2 = \mathit{Length}(\texttt{"http://"})$	$i_2 = 7$
$s_3 = Substring_2(s_1, i_2, i_1 - i_2)$	$\neg(i_2 < 0 \vee i_1 - i_2 < 0 \vee i_2 + i_1 - i_2 > \ell(s_1)) \wedge$ $\ell(s_3) = i_1 - i_2 \wedge \ell(s_1) \geq i_2 + i_1 - i_2$
$\mathit{StartsWith}(s_3, \texttt{"www."})$	$\ell(s_3) \geq 4$
$i_3 = \mathit{Length}(\texttt{"www."})$	$i_3 = 4$
$s_4 = \mathit{Substring}_1(s_3, i_3)$	$\neg(i_3 < 0 \vee i_3 > \ell(s_3)) \wedge \ell(s_4) = \ell(s_3) - i_3$
$s_4 = \texttt{"live.com"}$	$\ell(s_4) = 8$

Fig. 2. Integer abstraction of the example program

Fixed-length constraint satisfaction problems. A constraint C is said to be *fixed-length* if for every string variable s occurring in C, the constraint also contains a literal of the form $\ell(s) = i$, where i is an integer constant. One can easily note that in this case for every solution of C the length of s is i.

Consider any flat constraint C and its integer abstraction I. If I is unsolvable, then C has no solution either, Let us now take any value assignment v that solves I and consider the constraint C' obtained by adding to C all constraints of the form $\ell(s) = v(i)$, where s is a string variable occurring in C and i the integer variable denoting the length of s. One can immediately see that C' is a fixed-length constraint and that every solution to C' is also a solution to C. Our next observation is that the satisfiability problem for fixed-length constraints is decidable by Theorem 1.

6 Implementation and integration with an SMT solver

In Pex, Strings are represented as an abstract type String. We associate the predicate $\texttt{null}(s)$ that takes a string s and is true if the string represented by s is a null pointer.

The function `length`(s) results in the length of s, and the function `chars`(s) results in an array, whose domain consists of 32-bit bit-vectors and the range comprises of unicode characters (16-bit bit-vectors).

Building abstract execution paths All string library functions have implementations in Microsoft's .NET base class library, but many of these are in native code and therefore not in the scope of what Pex can analyze (Pex only analyzes .NET code). Pex therefore contains straightforward implementations of each of the string library functions written in C#. We show the implementation of the IndexOf function as an example below.

```
public static int IndexOf(string text, string key, int start, int count)
{
    if (text == null) throw new NullReferenceException();
    if (key == null) throw new ArgumentNullException();
    if (start < 0 | count < 0 | start + count < 0 |
        start + count > text.Length)
        throw new ArgumentOutOfRangeException();
    if (key.Length == 0) return start;
    if (count < key.Length) return -1;
    return IndexOfC(text, key, start, count);
}

private static int IndexOfC(string text, string key, int start, int count)
{
   int end = start + count - key.Length + 1;
   for (int i = start; i < end; i++) {
        bool b = true;
        for (int j = 0; b && j < count; j++)
            b &= text[i + j] == key[j];
        if (b) return i;
   }
   return -1;
}
```

The implementation contains two parts, the preamble within the body of IndexOf is a straight-line code sequence that checks for exception conditions and boundary values. These checks include the conditions listed in Table 2 and parts from the abstraction of Table 4 that cover also the concrete case. Then, the portion of the string function that we wish to abstract is encoded in IndexOfC.

At this point we can run standard dynamic symbolic execution with the string library functions by using the instructions within IndexOf and IndexOfC for both the concrete and symbolic execution. Our aim is however to abstract the part of IndexOfC into core string constraints. For this purpose we introduce the uninterpreted function IndexOfA. When executing IndexOfC in dynamic symbolic execution, we do not add constraints to the path condition, but set the symbolic result to $\mathsf{IndexOfA}(text, key, start, count)$.

Functions, such as Concat and Substring, do not depend on the string tokens. Pex encodes such functions using two primitives, Shift and Fuse, axiomatized by $\mathsf{Shift}(a,i)[j] \simeq a[i+j]$ and $\mathsf{Fuse}(a,i,b)[j] \simeq \textbf{if } j < i \textbf{ then } a[j] \textbf{ else } b[j]$. We can then replace $\mathsf{chars}(\mathsf{ConcatA}(a,b))$ by $\mathsf{Fuse}(\mathsf{chars}(a), \mathsf{length}(a), \mathsf{Shift}(\mathsf{chars}(b), -\mathsf{length}(a)))$, and $\mathsf{chars}(\mathsf{SubstringA}(a,i,j))$ by $\mathsf{Shift}(\mathsf{chars}(a), i)$.

Solving string constraints For each execution path, we get a path condition and perform multiple queries to Pex's SMT solver, Z3:

Phase 1. Assert the path condition π, the axioms for Fuse and Shift, and the axiom $\forall s \,.\, 0 \leq \mathsf{length}(s) < \mathcal{U}$, where $\mathcal{U}$ is a fresh constant. If the constraints are unsatisfiable, then fail; otherwise enter the second phase.

Phase 2. We are given a path constraint π that is satisfiable with respect to partial unfoldings of the supplied axioms. Find the smallest power of two for $\mathcal{U}$ such that the constraints have a model. Set $\mathcal{N} \leftarrow 0$.

1. Extract values from the model that suffice to create a finite unfolding of all quantifiers used in Table 3 for the functions: IndexOf, LastIndexOf, Contains, Compare, and Equals. Thus, we assert the definitions of these functions replacing all quantifiers of the form $(\forall i \in [a \ldots b])\varphi(i)$ with a finite set of assertions $\varphi(v(a)), \ldots, \varphi(v(b))$.
2. Instantiate the definitions of Shift and Fuse[3].
3. If the constraints are satisfiable, return the current model.
4. Fail if $\mathcal{N}$ or $\mathcal{U}$ exceed pre-configured bounds.
5. Otherwise, undo the assertions from steps 1 and 2 and force a solution with increased lengths by asserting $\Sigma_i \ell(s_i) > \Sigma_i v(\ell(s_i))$ where $\ell(s_i)$ occur in π.
6. If the new constraints are unsatisfiable, then fail.
7. Otherwise, repeat step 1 with $\mathcal{U} \leftarrow 2 \cdot \mathcal{U}, \mathcal{N} \leftarrow \mathcal{N} + 1$,

Note that step 5 can prevent exploring models where string lengths add up to the previous value, but are re-distributed in a different way. While $\mathcal{N}$ and $\mathcal{U}$ impose limits on which models are explored, it is the case that our implementation in Pex makes implicit use of properties of the bit-vector solver in Z3: Our solver tends to generate models with bit-vectors assigned to as large values as possible, so the progression of lengths tends to grow in proportion to $\mathcal{U}$.

To distribute length increases fairly among strings, Pex furthermore excludes strings whose length was increased recently, unless this exclusion would cause the constraints to become unsatisfiable.

7 Experiments

We applied Pex on the EasyChair[4] query given in Figure 1 using different search strategies to flip branches of already discovered execution paths (which includes unrolling loops): breadth-first (BFS), depth-first (DFS), flipping of Random branches, and Pex' default strategy [18] (which combines several heuristics). We compare those strategies with a *partial* implementation of the algorithm described above, where only Concat, Substring, Remove and Insert are abstracted, and the fully *abstract* version of the algorithm. We set the bounds of $\mathcal{U}$ at 2^{12}, and $\mathcal{N}$ at 3. Table 5 shows the results: BFS, DFS and Random

mode	time/s	paths
BFS	7.90	214
DFS	3.65	51
Random	8.73	196
Default	0.83	35
Partial	0.61	30
Abstract	1.02	19

Table 5. Evaluation of string solver on EasyChair

[3] While necessary for completeness, this step has not had any effect in our experiments.

[4] All experiments were performed with a Intel Core 2 CPU T7400 @ 2.16 Ghz, 4GB RAM.

c	1		2		3		4		5		6	
mode	time	paths	time	paths	time	paths	time	paths	time	paths	time	paths
BFS	0.14	5	4.50	140	73.34	1432	960.08	7727	timeout		timeout	
Random	0.26	7	0.42	9	4.93	57	16.70	108	40.55	199	154.36	541
Default	0.23	6	0.62	14	7.03	78	7.76	80	8.81	82	193.49	637
Abstract	0.15	6	0.33	8	0.99	10	2.02	12	4.44	14	6.67	16

Table 6. Evaluation of string solver on IndexOf progression

search performed clearly worse than the default and abstract strategies. The partial abstraction resulted in the fastest run-time and fewer explored paths than the default exploration strategy, while complete abstraction required exploring fewer paths, but took slightly more overall time.

A different example that highlights the effectiveness of the abstraction can be constructed by creating a program test with a conjunction of the form $s.\texttt{IndexOf}(s_1)! = -1\&\&s.\texttt{IndexOf}(s_2)! = -1\&\&\ldots$, where $s_1, s_2, \ldots, s_c$ are different non-overlapping strings with a long common prefix. The goal is to synthesize a string s that contains all the substrings $s_1, s_2, \ldots s_c$. Table 6 summarizes the results of trying a progression of $c = 1, \ldots, 6$ such conjuncts. DFS search, not shown, does not even manage to explore a single path, other strategies are also inferior to abstraction.

More examples are available at http://research.microsoft.com/Pex/Benchmarks/Strings/paper.aspx.

8 Conclusion

We presented a two-tier approach for generating finite models of path constraints for string-manipulating programs. Our approach views constraints from string libraries as extensions of the word equation problem and we identified decidable, undecidable, and open problems in the context of word equations. Our approach was integrated with the dynamic symbolic execution engine Pex.

Related Work. In the context of symbolic execution of programs, string abstractions has been recently studied by Ruan et.al. [14], where an approach based on a first-order encoding of string functions is proposed. They study C programs where strings are zero-terminated arrays whose lengths are bounded by constants. The first-order quantifiers can therefore be finitely unfolded and decided using a solver for linear arithmetic and assignments. Shannon et.al. [16] use automata-based representations for abstracting strings during symbolic execution of Java programs. They handle a few core methods in the java.lang.String class, and some other related classes. They integrate a numeric constraint solver, but apparently in a partial way. For example, string methods which return integers, such as IndexOf, cause case-splits over all possible return values within certain bounds. Automata-based methods have been pursued in the context of static analysis by Christensen et.al. [5], where automata, using the Mohri-Nederhof algorithm, represent over-approximations of possible string values. A motivation for the work is *SQL injection attacks*. The same motivation also inspired Fu et.al. [6]. They first solve Boolean and integer constraints to obtain a model and then proceed to solving string constraints by using the obtained model to build automata for the string constraints. To our knowledge, the mentioned automata-based methods require case analysis outside of their calls to their constraint solvers and automaton construction phases. In our framework, case analysis is integrated with the constraint solver pass.

Future work. A plethora of future work is possible in the context of exploring string manipulating programs. *Regular expressions* are often used by the discriminating programmer to accomplish string manipulation. In particular, our running EasyChair example can be directly encoded using a regular expression. But real regular expression libraries can encode side-effects and non-regular properties. Can such extensions be handled by methods presented here? A different direction is to extend array property fragments [2, 10] to handle common string queries. We would also like to use information encoded in the core language to control the constraint solver programmatically. For example, one could alternate quantifier unfolding with solving for the bounds.

Acknowledgments. We thank Wolfram Schulte for numerous early stage discussions and Yuri Matiyasevich for his help finding related work on word equations.

References

1. S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *Proc. of TACAS'08*, volume 4963 of *LNCS*, pages 367–381. Springer, 2008.
2. A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In E. A. Emerson and K. S. Namjoshi, editors, *VMCAI*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.
3. J. R. Büchi and S. Senger. Definability in the existential theory of concatenation. *Zeitschrift fur Mathematische Logik und Grundlagen der Mathematik*, 1988.
4. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. In *CCS*, pages 322–335, New York, NY, USA, 2006. ACM Press.
5. A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *SAS 03, volume 2694 of LNCS*, pages 1–18. Springer-Verlag, 2003.
6. X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao. A Static Analysis Framework For Detecting SQL Injection Vulnerabilities. In *COMPSAC*, pages 87–96, 2007.
7. P. Godefroid. Compositional dynamic test generation. In *Proc. of POPL'07*, pages 47–54, New York, NY, USA, 2007. ACM Press.
8. P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. *SIGPLAN Notices*, 40(6):213–223, 2005.
9. P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Proceedings of NDSS'08 (Network and Distributed Systems Security)*, pages 151–166, 2008.
10. P. Habermehl, R. Iosif, and T. Vojnar. What else is decidable about integer arrays? In R. M. Amadio, editor, *FoSSaCS*, volume 4962 of *LNCS*, pages 474–489. Springer, 2008.
11. B. Khoussainov, A. Nies, S. Rubin, and F. Stephan. Automatic structures: Richness and limitations. In *LICS*, pages 44–53, 2004.
12. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
13. Y. Matiyasevich. Word Equations, Fibonacci Numbers, and Hilbert's Tenth problem. In *Workshop on Fibonacci Words*, volume 43, pages 36–39, 2007.
14. H. Ruan, J. Zhang, and J. Yan. Test Data Generation for C Programs with String-Handling Functions. *Theoretical Aspects of Software Engineering*, 0:219–226, 2008.
15. K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *CAV*, pages 419–423, 2006.
16. D. Shannon, S. Hajra, A. Lee, D. Zhan, and S. Khurshid. Abstracting symbolic execution with string analysis. In *TAICPART-MUTATION*, pages 13–22, Washington, DC, USA, 2007.

17. N. Tillmann and J. de Halleux. Pex - white box test generation for .NET. In *Proc. of Tests and Proofs (TAP'08)*, volume 4966 of *LNCS*, pages 134–153, Prato, Italy, April 2008. Springer.
18. T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. Technical Report MSR-TR-2008-123, Microsoft, 2008.