String Constraints for Verification

Parosh Aziz Abdulla[1], Mohamed Faouzi Atig[1], Yu-Fang Chen[2], Lukáš Holík[3], Ahmed Rezine[4], Philipp Rümmer[1], and Jari Stenman[1]

[1] Department of Information Technology, Uppsala University, Sweden
[2] Institute of Information Science, Academia Sinica, Taiwan
[3] Faculty of Information Technology, Brno University of Technology, Czech Republic
[4] Department of Computer and Information Science, Linköping University, Sweden

Abstract. We present a decision procedure for a logic that combines: (i) word equations over string variables denoting words of arbitrary lengths, together with (ii) constraints on the lengths of the words, and on (iii) the regular languages to which they belong. The problem in its generality is still open. Our procedure is sound for the general logic. It is complete for a particularly rich fragment that restricts the form in which word equations are written. The logic for which our procedure is sound and complete is strictly more expressive than any existing procedure for handling string variables denoting words of arbitrary lengths. We provided a prototype integration of our decision procedure in a CEGAR-based model checker for the analysis of programs encoded as Horn clauses. Our tool is able to automatically establish the correctness of several programs that are beyond the reach of existing methods.

1 Introduction

Software model checking is an active research area that has witnessed a remarkable success in the past decades [16, 9]. Mature model checking tools are already used in industrial applications [2]. One main reason for this success is that such tools leverage on the latest developments in SMT technology [5, 8, 3] in order to reason about symbolic representations of different data types in programs. On the other hand, this dependence also means that model checking tools are inherently limited by the data types that can be handled by the underlying SMT solver. A data type for which satisfying decision procedures have been missing is that of *Strings*. Our work proposes a rich string logic together with a decision procedure targeting model checking applications.

String data types are present in all conventional programming and scripting languages. In fact, it is impossible to capture the essence of many programs, for instance in database and web applications, without the ability to precisely represent and reason about string data types. The control flow of programs can depend on the words denoted by the string variables, on their lengths, or on the regular languages to which they belong. For example, a program allowing users to choose a username and a password may require the password to be of a minimal length, to be different from the username, and to be free from invalid characters. Reasoning about such constraints is also crucial when verifying that database and web applications are free from SQL injections and other security vulnerabilities.

Existing solvers for programs manipulating string variables and their lengths are either unsound, not expressive enough, or lack the ability to provide counter examples. Many solvers [10, 25, 26] are unsound since they assume an a priori fixed upper bound on the length of the possible words. Others [10, 18, 28] are not expressive enough as they do not handle word equations, length constraints or membership predicates. Such works are mostly aimed at performing symbolic executions, i.e. establishing feasibility of paths in the program. The solver in [27] performs sound over-approximations but without supplying counter examples in case the verification fails. In contrast, our decision procedure specifically targets model checking applications. In fact, we use it in a prototype model checker in order to automatically establish program correctness for several examples.

Our decision procedure establishes satisfiability of formulae written as Boolean combinations of: (i) word (dis)equations such as $(a \cdot u = v \cdot b)$ or $(a \cdot u \neq v \cdot b)$, where a, b are letters and u, v are string variables denoting words of arbitrary lengths, (ii) length constraints such as $(|u| = |v| + 1)$, where $|u|$ refers to the length of the word denoted by string variable u, and (iii) predicates representing membership in regular expressions, e.g., $u \in c \cdot (a + b)^*$. Each of these predicates can be crucial for capturing the behavior and establishing the correctness of a string-manipulating program (cf. the small program in Section 2). The analysis is not trivial as it needs to capture subtle interactions between different types of predicates. For instance, the formulae $\phi_1 = (a \cdot u = v \cdot b) \wedge (|u| = |v| + 1)$ and $\phi_2 = (a \cdot u = v \cdot b) \wedge v \in c \cdot (a + b)^*$ are unsatisfiable, i.e., there is no possible assignment of words to u and v that makes the conjunctions evaluate to true. To capture this, the analysis needs to propagate facts from one type of predicates to another (e.g., in ϕ_1 the analysis deduces from $(a \cdot u = v \cdot b)$ that $|u| = |v|$ which results in an unsatisfiable formula $(|u| = |v| \wedge |u| = |v| + 1)$). The general decidability problem is still open. We guarantee termination of our decision procedure for a fragment of the full logic that includes the three types of predicates. The fragment we consider is rich enough to capture all the practical examples we have encountered.

We have integrated our decision procedure in a prototype model checker and used it to verify properties of implementations of common string manipulating functions such as the Hamming and Levenshtein distances. Predicates required for verification can be provided by hand; to achieve automation, in addition we propose a constraint-based interpolation procedure for regular word constraints. In combination with our decision procedure for words, this enables us to automatically analyse programs that are currently beyond the reach of state-of-the-art software model checkers.

Related Work. The pioneering work by Makanine [20] proposed a decision procedure for word equations (i.e. Boolean combinations of (dis)equalities) where the variables can denote words of arbitrary lengths. The decidability problem is already open [4] when word equations are combined with length constraints of the form $|u| = |v|$. The logic we are considering adds predicates representing membership in regular languages to word equations and length constraints. This means that decidability is still an open problem. A contribution of our work is the definition of a rich sublogic for which we guarantee the termination of our sound decision procedure.

In a work close to ours, the authors in [11] show decidability of a logic that is strictly weaker than the one for which we guarantee termination of our decision procedure. For instance, in [11], membership predicates are allowed only under the assumption that no string variables can appear in the right hand sides of the equality predicates. This severely restricts the expressivity of the logic. In [28], the authors augment the Z3 [7] SMT solver in order to handle word equations with length constraints. However, they do not support regular membership predicates. In our experience, these are crucial during model checking based verification.

Finally, in addition to considering more general equations, our work comes with an interpolation based verification technique adapted for string programs. Notice that neither of [11, 28] can establish correctness of programs with loops.

Outline. In the next section, we use a simple program to illustrate our approach. In Section 3 we introduce a logic for word equations with arithmetic and regular constraints, and then describe in Section 4 a procedure for deciding satisfiability of formulae in the logic. In Section 5 we define a class formulae for which we guarantee the termination of our decision procedure. We describe the verification procedure in Section 6 and the implementation effort in Section 7. Finally in Section 8 we give some conclusions and directions for future work.

2 A Simple Example

In this section, we use the simple program listed in Fig. 1 to give a flavor of our verification approach. The listing makes use of features that are common in string manipulating programs. We will argue that establishing correctness for such programs requires: (i) the ability to refer to string variables of arbitrary lengths, (ii) the ability to express combinations of constraints, like that the words denoted by the variables belong to regular expressions, that their lengths obey arithmetic inequalities, or that the words themselves are solutions to word equations, and (iii) the ability for a decision procedure to precisely capture the subtle interaction between the different kinds of involved constraints.

In the program of Fig. 1, a string variable `s` is initialized with the empty word. A loop is then executed an arbitrary number of times. At each iteration of the loop, the instruction `s= 'a' + s + 'b'` appends the letter `'a'` at the beginning of variable `s` and the letter `'b'` at its end. After the loop, the program asserts that that `s` does not have the word `'ba'` as a substring (denoted by `!s.contains('ba')`, and that its length (denoted by `s.length()`) is even.

Observe that the string variable `s` does not assume a maximal length. Any verification procedure that requires an a priori fixed bound on the length of the

```
// $Pre = (true)$
String s= '';
// $P_1 = (s \in \epsilon)$
while(*){
     // $P_2 = (s = u \cdot v \wedge u \in a^* \wedge v \in b^* \wedge |u| = |v|)$
     s= 'a' + s + 'b';
}
// $P_3 = P_2$
assert(!s.contains('ba') && (s.length() % 2) == 0);
// $Post = P_3$
```

Fig. 1. A simple program manipulating a string variable s. Our logic allows to precisely capture the word equations, membership predicates and length constraints that are required for validating the assertion is never violated. Our decision procedure can then automatically validate the required verification conditions described in Fig. 2.

$vc_1 : post(Pre, \mathtt{s} = "") \implies P_1$
$vc_2 : P_1 \implies P_2$
$vc_3 : post(P_2, \mathtt{s} = "\mathtt{a}" \cdot \mathtt{s} \cdot "\mathtt{b}") \implies P_2$
$vc_4 : P_2 \implies P_3$
$vc_5 : post(P_3,$ `assume(s.contains("ba") || !(s.length()%2 ==0)))` $\implies false$
$vc_6 : post(P_3,$ `assume(!s.contains("ba") && (s.length()%2 ==0)))` $\implies Post$

Fig. 2. Verification conditions for the simple program of Fig. 1.

string variables is necessarily unsound and will fail to establish correctness for the program.

Moreover, establishing correctness requires the ability to express and to reason about predicates such as those mentioned in the comments of the code in Fig. 1. For instance, the loop invariant P_2 states that: (i) the variable s denotes a finite word w_s of arbitrary length, (ii) that w_s equals the concatenation of two words w_u and w_v, (iii) that $w_u \in a^*$ and $w_v \in b^*$, and (iv) that the length $|w_u|$ of word w_u equals the length $|w_v|$ of word w_v.

Using the predicates in Fig. 1, we can formulate program correctness in terms of the validity of each of the implications listed in Fig. 2. For instance, validity of the verification condition vc_5 amounts to showing that $\neg vc_5 = (s = u \cdot v \wedge u \in a^* \wedge v \in b^* \wedge |u| = |v|) \wedge (s = s_1 \cdot b \cdot a \cdot s_2 \vee \neg(|s| = 2n))$ is unsatisfiable. To establish this result, our decision procedure generates the two proof obligations $\neg vc_{51} : (s = u \cdot v \wedge u \in a^* \wedge v \in b^* \wedge |u| = |v| \wedge s = s_1 \cdot b \cdot a \cdot s_2)$ and $\neg vc_{52} : (s = u \cdot v \wedge u \in a^* \wedge v \in b^* \wedge |u| = |v| \wedge \neg(|s| = 2n))$.

In order to check vc_{51}, the procedure symbolically matches all the possible ways in which a word denoted by $u \cdot v$ can also be denoted by $s_1 \cdot b \cdot a \cdot s_2$. For instance, $u = s_1 \cdot b \wedge v = a \cdot s_2$ is one possible matching. In order to be able to show unsatisfiability, the decision procedure has to also consider the other possible matchings. For instance, the case where the word denoted by u is a strict prefix of the one denoted by s_1 has also to be considered. For this reason, the matching process might trigger new matchings. In general, there is no guaranty that the sequence of generated matchings will terminate. However, we show that this sequence terminates for an expressive fragment of the logic. This fragment includes the predicates of mentioned in this section and all predicates we encountered in practical programs, The procedure then checks satisfiability of each such a matching. For instance, the matching $u = s_1 \cdot b \wedge v = a \cdot s_2$ is shown to be unsatisfiable due the the membership predicate $v \in b^*$. In fact our procedure automatically proves that $\neg v_{51}$ is not satisfiable after checking all possible matchings.

So for $\neg vc_5$ to be satisfiable, $\neg vc_{52}$ needs to be satisfiable. Our procedure deduces that this would imply that $|u| = |v| \wedge \neg(|u| + |v| = 2n)$ is satisfiable. We leverage on existing standard decision procedures for linear arithmetic in order to show that this is not the case. Hence $\neg vc_5$ is unsatisfiable and vc_5 is valid. For this example, and those we report on in Section 6, our procedure can establish correctness fully automatically given the required predicates.

Observe that establishing validity requires the ability to capture interactions among the different types of predicates. For instance, establishing validity of vc_5 involves the ability to combine the word equations $(s = u \cdot v \wedge s = s_1 \cdot b \cdot a \cdot s_2)$ with the membership predicates $(u \in a^* \wedge v \in b^*)$ for vc_{51}, and with the length constraints $(|u| = |v| \wedge \neg(|s| = 2n))$ for vc_{52}. Capturing such interactions is crucial for establishing correctness and for eliminating false positives.

3 Defining the String Logic $\mathcal{E}_{e,r,l}$

In this section we introduce a logic, which we call $\mathcal{E}_{e,r,l}$, for word equations with arithmetic and regular constraints. We assume a finite alphabet Σ and write Σ^* to mean the set of finite words over Σ. We work with a set U of string variables denoting words in Σ^* and write $\mathcal{Z}$ for the set of integer numbers.

Syntax. We let variables u, v range over the set U. We write $|u|$ to mean the length of the word denoted by variable u, k to mean an integer in $\mathcal{Z}$, c to mean a letter in Σ and w to mean a word in Σ^*. The syntax of formulae in $\mathcal{E}_{e,r,l}$ is defined as follows:

$$\begin{array}{rcll}
\phi & ::= & \phi \wedge \phi \mid \neg\phi \mid \varphi_e \mid \varphi_l \mid \varphi_r & \text{formulae} \\
\varphi_e & ::= & tr = tr \mid tr \neq tr & \text{(dis)equalities} \\
\varphi_l & ::= & e \leq e & \text{arithmetic inequalities} \\
\varphi_r & ::= & tr \in \mathcal{R} & \text{membership predicates} \\
tr & ::= & \epsilon \mid c \mid u \mid tr \cdot tr & \text{terms} \\
\mathcal{R} & ::= & \emptyset \mid \epsilon \mid c \mid w \mid \mathcal{R} \cdot \mathcal{R} \mid \mathcal{R} + \mathcal{R} \mid \mathcal{R} \cap \mathcal{R} \mid \mathcal{R}^C \mid \mathcal{R}^* & \text{regular expressions} \\
e & ::= & k \mid |tr| \mid k * e \mid e + e & \text{integer expressions}
\end{array}$$

Assume variables $\{u_i\}_{i=1}^n$, terms $\{tr_i\}_{i=1}^n$ and integer expressions $\{e_i\}_{i=1}^n$. We write $\phi[u_1/tr_1] \ldots [u_n/tr_n]$ (resp. $\phi[|u_1|/e_1] \ldots [|u_n|/e_n]$) to mean the formula obtained by syntactically substituting in ϕ each occurence of u_i by term tr_i (resp. each occurence of $|u_i|$ by expression e_i). Such a substitution is said to be well defined if no variable u_i (resp. expression e_i) appears in any tr_i (resp. e_i).

The subscripts e, r, l in $\mathcal{E}_{e,r,l}$ respectively stand for word equations, membership predicates in regular expressions (membership predicates for short) and length constraints. The set of word variables appearing in a term is defined as follows: $Vars(\epsilon) = \emptyset$, $Vars(c) = \emptyset$, $Vars(u) = \{u\}$ and $Vars(tr_1 \cdot tr_2) = Vars(tr_1) \cup Vars(tr_2)$.

Semantics. The semantics of $\mathcal{E}_{e,r,l}$ is mostly standard. We describe it using a mapping η (called *interpretation*) that assigns words in Σ^* to string variables in U. We extend η to terms as follows: $\eta(\epsilon) = \epsilon$, $\eta(c) = c$ and $\eta(tr_1.tr_2) = \eta(tr_1).\eta(tr_2)$. Every regular expression $\mathcal{R}$ is evaluated to the language $\mathcal{L}(\mathcal{R})$ it represents. Given an interpretation η, we define another mapping β_η that associates a number in $\mathcal{Z}$ to integer expressions as follows: $\beta_\eta(k) = k$, $\beta_\eta(|u|) = |\eta(u)|$, $\beta_\eta(|tr|) = |\eta(tr)|$, $\beta_\eta(k * e) = k * \beta_\eta(e)$, and $\beta_\eta(e_1 + e_2) = \beta_\eta(e_1) + \beta(e_2)$. A formula in $\mathcal{E}_{e,r,l}$ is then evaluated to a value in $\{ff, tt\}$ as follows:

$$\begin{array}{rcll}
val_\eta(\phi_1 \wedge \phi_2) & = & tt & \text{iff} \quad val_\eta(\phi_1) = tt \text{ and } val_\eta(\phi_2) = tt \\
val_\eta(\neg\phi_1) & = & tt & \text{iff} \quad val_\eta(\phi_1) = ff \\
val_\eta(tr \in \mathcal{R}) & = & tt & \text{iff} \quad \eta(tr) \in \mathcal{L}(\mathcal{R}) \\
val_\eta(tr_1 = tr_2) & = & tt & \text{iff} \quad \eta(tr_1) = \eta(tr_2) \\
val_\eta(tr_1 \neq tr_2) & = & tt & \text{iff} \quad \neg(\eta(tr_1) = \eta(tr_2)) \\
val_\eta(e_1 \leq e_2) & = & tt & \text{iff} \quad \beta_\eta(e_1) \leq \beta_\eta(e_2)
\end{array}$$

A formula ϕ is said to be *satisfiable* if there is an interpretation η such that $val_\eta(\phi) = tt$. It is said to be *unsatisfiable* otherwise.

4 Deciding Satisfiability

In this Section we describe our procedure for deciding satisfiability of formulae in the logic $\mathcal{E}_{e,r,l}$ of Section 3. Given a formula ϕ, we build a proof tree rooted at ϕ by repeatedly applying the inference rules introduced in this Section. We can assume, without loss of generality, that the formula is given in Disjunctive Normal Form. An inference rule is of the form:

$$\textsc{Name} : \frac{B_1 \ B_2 \ \ldots \ B_n}{A} \ cond$$

In this inference rule, Name is the name of the rule, *cond* is a prerequisite on A for the application of the rule, B_1 B_2 ... B_n are called premises and A is called the conclusion of the rule. The premises and conclusion are formulae in $\mathcal{E}_{e,r,l}$. Each application consumes a conclusion and produces the set of premises. The inference rule is said to be *sound* if the satisfiability of the conclusion implies the satisfiability of one of the premises. It is said to be *locally complete* if the satisfiability of one of the premises implies the satisfiability of the conclusion. If all inference rules are locally complete, and if ϕ or one of the produced premises turns out to be satisfiable, then ϕ is also satisfiable. If all the inference rules are sound and none of the produced premises is satisfiable, then ϕ is also unsatisfiable.

We organize the inference rules in four groups. We use the rules of the first group to eliminate disequalities. The rules of the second group are used to simplify equalities. The rules of the third group are used to eliminate membership predicates. The rules of the last group are used to propagate length constraints. In addition, we assume standard decision procedures [3] for integer arithmetic.

4.1 Removing Disequalities

We use rules Not-Eq and Diseq-Split in order to eliminate disequalities. In the first rule, we establish that $tr \neq tr \wedge \phi$ is not satisfiable and close this branch of the proof. In the second rule Diseq-Split, we eliminate disequalities involving arbitrary terms. For this, we make use of the fact that the alphabet Σ is finite and replace any disequality with a finite set of equalities. More precisely, assume a formula $tr \neq tr' \wedge \phi$ in $\mathcal{E}_{e,r,l}$. We observe that the disequality $tr \neq tr'$ holds iff the words w_{tr} and $w_{tr'}$ denoted by the terms tr and tr' are different. This corresponds to one of three cases. Assume three fresh variables u, v and v'. In the first case, the words w_{tr} and $w_{tr'}$ contain different letters $c \neq c'$ after a common prefix w_u. They are written as the concatenations $w_u \cdot c \cdot w_v$ and $w_u \cdot c' \cdot w'_v$ respectively. We capture this case using the set $\textsc{Split}_{\textsc{Diseq-Split}} = \{tr = u \cdot c \cdot v \wedge tr' = u \cdot c' \cdot v' \wedge \phi \mid c, c' \in \Sigma \text{ and } c \neq c'\}$. In the second case, the word $w_{tr'} = w_u$ is a strict prefix of $w_{tr} = w_u \cdot c \cdot w_v$. We capture this with $\textsc{Split}'_{\textsc{Diseq-Split}} = \{tr = u \cdot c \cdot v \wedge tr' = u \wedge \phi \mid c \in \Sigma\}$. In the third case, the word $w_{tr} = w_u$ is a strict prefix of $w_{tr'} = w_u \cdot c' \cdot w'_v$, and we capture this case using the set $\textsc{Split}''_{\textsc{Diseq-Split}} = \{tr = u \wedge tr' = u \cdot c \cdot v' \wedge \phi \mid c \in \Sigma\}$.

$$\textsc{Diseq-Split}: \frac{\textsc{Split}_{\textsc{Diseq-Split}} \cup \textsc{Split}'_{\textsc{Diseq-Split}} \cup \textsc{Split}''_{\textsc{Diseq-Split}}}{tr \neq tr' \wedge \phi}$$

$$\textsc{Not-Eq}: \frac{*}{tr \neq tr \wedge \phi} \qquad\qquad \textsc{Eq}: \frac{\phi}{tr = tr \wedge \phi}$$

4.2 Simplifying Equalities

We introduce rules Eq, A-Eq-Var, C-Eq-Var, and Eq-Word to manipulate equalities. Rule applications take into account symmetry of the equality operator (i.e., if a rule can apply to $w \cdot tr_1 = tr_2 \wedge \phi$ then it can also apply to $tr_2 = w \cdot tr_1 \wedge \phi$). Rule Eq eliminates trivial equalities of the form $tr = tr'$.

Rules A-Eq-Var and C-Eq-Var eliminate variable u from the equality $u \cdot tr_1 = tr_2 \wedge \phi$. The first rule considers the case where the formula $u \cdot tr_1 = tr_2 \wedge \phi$ is $\underline{\text{a}}$cyclic (hence the A in A-Eq-Var). Acyclicity is defined in Section 5. It is a simple syntactic condition on formulae that guarantees abscense of recursive dependencies among string variables. For instance, it forbidds the presence of variable u in term tr_2. The second rule captures the case where the formula is not acyclic (hence variables may repeatedly appear on both sides of (dis)equations). Let w_u be some word denoted by u. For the equality to hold, w_u must also be denoted by some prefix of the word denoted by tr_2. There are two similar cases for both rules. In the first case, w_u is also denoted by a prefix tr_3 of tr_2. In the second case, w_u does not exactly match a prefix of the word denoted by tr_2. Instead, tr_2 can be written as $tr_3 \cdot v \cdot tr_4$ and the word w_u is written as the concatenation of two words, one that is denoted by tr_3 and another that is prefix of the word denoted by v.

$$\text{A-Eq-Var} : \frac{\textsc{Split}_{\text{A-Eq-Var}} \cup \textsc{Split}'_{\text{A-Eq-Var}}}{u \cdot tr_1 = tr_2 \wedge \phi} \ (u \cdot tr_1 = tr_2 \wedge \phi) \text{ is acyclic}$$

We capture both cases for rule A-Eq-Var using the two sets: $\textsc{Split}_{\text{A-Eq-Var}} = \{(tr_1 = tr_4 \wedge \phi)[u/tr_3] | tr_2 = tr_3 \cdot tr_4\}$, together with the set $\textsc{Split}'_{\text{A-Eq-Var}} = \{((tr_1 = v_2 \cdot tr_4 \wedge \phi)\,[u/tr_3 \cdot v_1])\,[v/v_1 \cdot v_2] | tr_2 = tr_3 \cdot v \cdot tr_4\}$. Observe the acyclicity condition guarantees the substitutions are well defined.

$$\text{C-Eq-Var} : \frac{\textsc{Split}_{\text{C-Eq-Var}} \cup \textsc{Split}'_{\text{C-Eq-Var}}}{u \cdot tr_1 = tr_2 \wedge \phi} \ (u \cdot tr_1 = tr_2 \wedge \phi) \text{ is not acyclic}$$

We define similar sets for the rule C-Eq-Var, namely: $\textsc{Split}_{\text{C-Eq-Var}} = \{(tr_1 = tr_4 \wedge u = tr_3 \wedge \phi) \mid tr_2 = tr_3 \cdot tr_4\}$ together with the set $\textsc{Split}'_{\text{C-Eq-Var}} = \{(tr_1 = v_2 \cdot tr_4 \wedge u = tr_3 \cdot v_1 \wedge v = v_1 \cdot v_2 \wedge \phi)\, | tr_2 = tr_3 \cdot v \cdot tr_4\}$. Observe that these definitions mimic the ones of $\textsc{Split}_{\text{C-Eq-Var}}$ and $\textsc{Split}_{\text{C-Eq-Var}}$, but do not use substitution to avoid introducing equalities. It is indeed meaningless to use substitution due to the recursive dependencies in cyclic formulae.

Rule Eq-Word eliminates the word w from the equality $w \cdot tr_1 = tr_2 \wedge \phi$. Again, we define: $\textsc{Split}_{\text{Eq-Word}} = \{(tr_3 \in w \wedge tr_4 = tr_1 \wedge \phi) | tr_2 = tr_3 \cdot tr_4\}$, and $\textsc{Split}'_{\text{Eq-Word}} = \{tr_3 \cdot v_1 \in w \wedge v_2 \cdot tr_4 = tr_1 \wedge \phi)[v/v_1 \cdot v_2] | tr_2 = tr_3 \cdot v \cdot tr_4\}$.

$$\text{Eq-Word} : \frac{\textsc{Split}_{\text{Eq-Word}} \cup \textsc{Split}'_{\text{Eq-Word}}}{w \cdot tr_1 = tr_2 \wedge \phi}$$

To simplify the presentation, we do not present suffix versions for rules A-Eq-Var, C-Eq-Var and Eq-Word. Such rules match suffixes instead of prefixes and simply mirror the rules described above.

4.3 Removing Membership Predicates

We use rules Reg-Neg, Memb, Not-Memb, Reg-Split and Lengths to simplify and eliminate membership predicates. We describe them below.

Rule Reg-Neg replaces the negation of a membership predicate in a regular expression $\mathcal{R}$ with a membership predicate in its complement $\mathcal{R}^C$.

$$\textsc{Reg-Neg} : \frac{tr \in \mathcal{R}^C \wedge \phi}{\neg(tr \in \mathcal{R}) \wedge \phi}$$

Rule MEMB eliminates the predicate $w \in \mathcal{R}$ in case the the word w belongs to the language $\mathcal{L}(\mathcal{R})$ of the regular expression $\mathcal{R}$. If w does not belong to $\mathcal{L}(\mathcal{R})$ then rule NOT-MEMB closes this branch of the proof.

$$\textsc{Memb} : \frac{\phi}{w \in \mathcal{R} \wedge \phi}\ w \in \mathcal{L}(\mathcal{R}) \qquad \textsc{Not-Memb} : \frac{*}{w \in \mathcal{R} \wedge \phi}\ w \notin \mathcal{L}(\mathcal{R})$$

Rule REG-SPLIT simplifies membership predicates of the form $tr \cdot tr' \in \mathcal{R}$. Given such a predicate, the rule replaces it with a disjunction $\bigvee_{i=1}^{n} \big(tr \in \mathcal{R}_i \wedge tr' \in \mathcal{R}'_i\big)$ where the set $\{(\mathcal{R}_i, \mathcal{R}'_i)\}_{i=1}^{n}$ is finite and only depends on the regular expression $\mathcal{R}$. To define this set, represent $\mathcal{L}(\mathcal{R})$ using some arbitrary but fixed finite automaton (S, s_0, δ, F). Assume $S = \{s_0, \ldots, s_n\}$. Choose the regular expressions $\mathcal{R}_i, \mathcal{R}'_i$ such that : (1) $\mathcal{R}_i$ has the same language as the automaton $(S, s_0, \delta, \{s_i\})$, and (2) $\mathcal{R}'_i$ has the same language as the automaton(S, s_i, δ, F). For any word $w_{tr} \cdot w_{tr'}$ denoted by $tr \cdot tr'$ and accepted by $\mathcal{R}$, there will be a state s_i in S such that w_{tr} is accepted by $\mathcal{R}_i$ and $w_{tr'}$ is accepted by $\mathcal{R}'_i$. Given a regular expression $\mathcal{R}$, we let $\mathcal{F}(\mathcal{R})$ denote the set $\{(\mathcal{R}_i, \mathcal{R}'_i)\}_{i=1}^{n}$ above.

$$\textsc{Reg-Split} : \frac{\{tr \in \mathcal{R}' \wedge tr' \in \mathcal{R}'' \wedge \phi \mid (\mathcal{R}', \mathcal{R}'') \in \mathcal{F}(\mathcal{R})\}}{tr \cdot tr' \in \mathcal{R} \wedge \phi}$$

Rule REG-LENG can only be applied in certain cases. To identify these cases, we define the condition $\Gamma(\phi, u)$ which states, given a formula ϕ and a variable u, that u is not used in any membership predicate or in any (dis)equation in ϕ. In other words, the condition states that if u occurs in ϕ then it occurs in a length predicates. The rule REG-LENG replaces, in one step, all the membership predicates $\{u \in \mathcal{R}_i\}_{i=1}^{n}$ with an arithmetic constraint $Len(\mathcal{R}_1 \cap \ldots \cap \mathcal{R}_m, u)$. This arithmetic constraint expresses that the length $|u|$ of variable u belongs to the semi linear set corresponding to the Parikh image of the intersection of all regular expressions $\{\mathcal{R}_i\}_{i=1}^{n}$ appearing in membership predicates of variable u. It is possible to determine a representation of this semi linear set by starting from a finite state automaton representing the intersection $\cap_i \mathcal{R}_i$ and replacing all letters with a unique arbitrary letter. The obtained automaton is determinized and the semi linear set is deduced from the length of the obtained lasso if any (notice that since the automaton is deterministic and its alphabet is a singelton, its from will be either a lasso or a simple path.) After this step, there will be no membership predicates involving variable u.

$$\textsc{Reg-Leng} : \frac{Len(\mathcal{R}_1 \cap \ldots \cap \mathcal{R}_m, u) \wedge \phi}{u \in \mathcal{R}_1 \wedge \ldots \wedge u \in \mathcal{R}_m \wedge \phi}\ \Gamma(\phi, u)$$

4.4 Propagating Term Lengths

The rule TERM-LENG is the only inference rule in the fourth group. It substitutes the expression $|tr| + |tr'|$ for every occurrence in ϕ of the expression $|tr \cdot tr'|$.

$$\text{TERM-LENG}: \frac{\phi[|tr \cdot tr'|/|tr| + |tr'|]}{\phi} \; |tr \cdot tr'| \text{ appears in } \phi$$

Lemma 1. *Inference rules defined in Section 4 are sound and locally complete.*

5 Completeness of the Decision Procedure

In this Section, we define a class formulae of *acyclic form* (we say a formula is in acyclic form, or acyclic for shor) for which the decision procedure in Section 4 is guaranteed to terminate. For simplicity, we assume that the formula is a conjunction of predicates and negated predicates.

Non-termination may be caused by an infinite chain of applications of the rules for removing equalities of Section 4.2 (such as C-EQ-VAR). Consider for instance the equality $u \cdot v = v \cdot u$. One of the cases generated within the disjunct $\text{SPLIT}_{\text{C-EQ-VAR}}$ of C-EQ-VAR is $v_1 \cdot v_2 = v_2 \cdot v_1$. This is the same as the original equality up to renaming of variables. In this case, the process of removing equalities clearly does not terminate. To prevent this, we will require that no variable can appear on both sides of an equality. The condition must hold for the initial formula and it must be preserved by an application of rules presented in Sections 4. Attention must be given to rules that modify equalities. Rules such as C-EQ-VAR involve substitution of a variable from one side of an equality by a term from the other side. We need to prevent *chains* of such substitutions that cause variables to appear on both sides of an equality. Acyclic formulae must also guarantee that the undesired cases cannot appear after a use of DISEQ-SPLIT of Section 4.1 that transforms a disequality to equalities. The following definition captures this formally.

Acyclic form. We associate to each formula ϕ an undirected labeled graph G_ϕ. We identify the nodes of G_ϕ with the string variables appearing in ϕ. Let $e : tr \approx tr'$ be a (dis)equality where $\approx \in \{=, \neq\}$. If tr has a variable u on the ith position and tr' has a variable v on the jth position, then G_ϕ connects u with v by an edge labeled by $e(u : i, v : j)$. The formula ϕ is said to be in an *acyclic form* if there are no loops in G_ϕ.

Notice that acyclic form also prevents repetition of a variable inside one side of a (dis)equality. This is needed in cases like $u \cdot u = v \cdot v$ where the rule C-EQ-VAR generates $v_1 = v_2 \cdot v_1 \cdot v_2$, with a variable v_1 on both sides of the equality, which is the situation which we wanted to prevent at the first place. The graph here contains four edges between u and v labeled by $e(u : i, v : j)$ where $i, j \in \{1, 2\}$ and hence it has a cycle.

Theorem 1. *The decision procedure of Section 4 terminates when applied to a formula in acyclic form.*

The proof of the Theorem can be found in the Appendix A. Together with Lemma 1, Theorem 1 implies that the decision procedure of Section 4 is sound and complete for formulae in the acyclic form.

Proving that there cannot be an infinite sequence of application of rules of Section 4 is done by showing that the application of each rule decreases measures defined as follows. The measure for most of the rules is natural: It is the number of disequalities (Diseq-Split), the number of appearances of constants (e.g, Eq-Word), the number of concatenations within membership predicates (Reg-Split), and the number of membership predicates (e.g, Reg-Length). The exception is the rule A-Eq-Var and its symmetrical variant where the definition of the decreasing measure is more intricate and it is only possible under the acyclicity assumption.

6 Complete Verification of String-Processing Programs

The analysis of string-processing programs has gained importance due to the increased use of string-based APIs and protocols, for instance in the context of databases and Web programming. Much of the existing work has focused on the detection of bugs or the synthesis of attacks; in contrast, the work presented in this paper primarily targets verification of *functional correctness.* The following sections outline how we use our logic $\mathcal{E}_{e,r,l}$ for this purpose. On the one hand, our solver is designed to handle the satisfiability checks needed when constructing finite abstractions of programs, with the help of predicate abstraction [12, 14] or Impact-style algorithms [21]; since $\mathcal{E}_{e,r,l}$ can express both length properties and regular expressions, it covers predicates sufficient for a wide range of verification applications. On the other hand, we propose a constraint-based Craig interpolation algorithm for the automatic refinement of program abstractions (Section 6.2), leading to a completeness result in the style of [17]. We represent programs in the framework of Horn clauses [22, 13], which make it easy to handle language features like recursion; however, our work is in no way restricted to this setting.

6.1 Horn Constraints with Strings

In our context, a *Horn clause* is a formula $H \leftarrow C \wedge B_1 \wedge \cdots \wedge B_n$ where C is a formula (constraint) in $\mathcal{E}_{e,r,l}$; each B_i is an application $p(t_1, \ldots, t_k)$ of a relation symbol $p \in \mathcal{R}$ to first-order terms; H is either an application $p(t_1, \ldots, t_k)$ of $p \in \mathcal{R}$ to first-order terms, or the constraint *false*. H is called the *head* of the clause, $C \wedge B_1 \wedge \cdots \wedge B_n$ the *body.* A set $\mathcal{HC}$ of Horn clauses is called *solvable* if there is an assignment that maps every n-ary relation symbol p to a word formula $C_p[x_1, \ldots, x_n]$ with n free variables, such that every clause in $\mathcal{HC}$ is valid. Since Horn clauses can capture properties such as initiation and consecution of invariants, programs can be encoded as sets of Horn clauses in such a way that the clauses are solvable if and only if the program is correct.

Example 1. The example from Section 2 is represented by the following set of Horn clauses, encoding constraints on the intermediate assertions Pre, P_1, P_2, P_3. Note that the clauses closely correspond to the verification conditions given in Fig. 2. Any solution of the Horn clauses represents a set of mutually inductive invariants, and witnesses correctness of the program.

$$\begin{aligned} Pre(s) &\leftarrow \mathit{true} \\ P_1(s') &\leftarrow s' = \epsilon \wedge Pre(s) \\ P_2(s) &\leftarrow P_1(s) \\ P_2(\text{"}a\text{"} \cdot s \cdot \text{"}b\text{"}) &\leftarrow P_2(s) \end{aligned} \qquad \begin{aligned} &P_3(s) \leftarrow P_2(s) \\ &\mathit{false} \leftarrow s \in (a|b)^* \cdot ba \cdot (a|b)^* \wedge P_3(s) \\ &\mathit{false} \leftarrow \forall k.\ 2k \neq |s| \wedge P_3(s) \end{aligned}$$

Algorithms to construct solutions of Horn clauses with the help of *predicate abstraction* have been proposed for instance in [13]; in this context, automatic solving is split into two main steps:1. the synthesis of *predicates* as building blocks for solutions, and 2. the construction of solutions as Boolean combinations of the predicates. The second step requires a solver to decide consistency of sets of predicates, as well as implication between predicates (a set of predicates implies some other predicate); our logic is designed for this purpose.

$\mathcal{E}_{e,r,l}$ covers a major part of the string operations commonly used in software programs; further operations can be encoded elegantly, including:

– *extraction of substring* v of length *len* from a string u, starting at position *pos*, which is defined by the formula:

$$u = p \cdot v \cdot s \wedge |v| = \mathit{len} \wedge |p| = \mathit{pos}$$

– *replacement* of the substring v by v', resulting in the new overall string u':

$$u = p \cdot v \cdot s \wedge u' = p \cdot v' \cdot s \wedge |v| = \mathit{len} \wedge |p| = \mathit{pos}$$

– similarly, *search* for the first occurrence of a string can be encoded, using either equations or regular expression constraints.

6.2 Constraint-Based Craig Interpolation

In order to synthesise new predicates for verification, we propose a constraint-based *Craig interpolation* algorithm [6]. We say that a formula $I[\bar{s}]$ is an interpolant of a conjunction $A[\bar{s}], B[\bar{s}]$ over common variables $\bar{s} = s_1, \ldots, s_n$ (and possibly including further local variables), if the conjunctions $A[\bar{s}] \wedge \neg I[\bar{s}]$ and $B[\bar{s}] \wedge I[\bar{s}]$ are unsatisfiable. In other words, an interpolant $I[\bar{s}]$ is an over-approximation of $A[\bar{s}]$ that is disjoint from $B[\bar{s}]$. It is well-known that interpolants are suitable candidates for predicates in software model checking; for a detailed account on the use of interpolants for solving Horn clauses, we refer the reader to [24].

Our interpolation procedure is shown in Alg. 1, and generates interpolants in the form of regular constraints separating $A[\bar{s}]$ and $B[\bar{s}]$. This means that interpolants are not arbitrary formulae in the logic $\mathcal{E}_{e,r,l}$, but are restricted to

Algorithm 1: Constraint-based interpolation of string formulae.

```
   Input: Interpolation problem $A[\bar{s}] \wedge B[\bar{s}]$ with common variables $\bar{s}$; bound $L$.
   Output: Interpolant $s_1|s_2|\cdots|s_n \in \mathcal{R}$; or result Inseparable.
1  $Aw \leftarrow \emptyset$;  $Bw \leftarrow \emptyset$;
2  while there is RE $\mathcal{R}$ of size $\leq L$ such that $Aw \subseteq \mathcal{L}(\mathcal{R})$ and $Aw \cap \mathcal{L}(\mathcal{R}) = \emptyset$ do
3      if $A[\bar{s}] \wedge \neg(s_1|s_2|\cdots|s_n \in \mathcal{R})$ is satisfiable with assignment $\eta$ then
4          $Aw \leftarrow Aw \cup \{\eta(s_1)|\cdots|\eta(s_n)\}$;
5      else if $B[\bar{s}] \wedge (s_1|s_2|\cdots|s_n \in \mathcal{R})$ is satisfiable with assignment $\eta$ then
6          $Bw \leftarrow Bw \cup \{\eta(s_1)|\cdots|\eta(s_n)\}$;
7      else
8          return $s_1|s_2|\cdots|s_n \in \mathcal{R}$;
9      end
10 end
11 return Inseparable;
```

the form $s_1|s_2|\cdots|s_n \in \mathcal{R}$, where "$|$" $\in \Sigma$ is a distinguished separating letter, and $\mathcal{R}$ is a regular expression. In addition, only interpolants up to a *bound* L are considered; L can limit, for instance, the length of the regular expression $\mathcal{R}$, or the number of states in a finite automaton representing $\mathcal{R}$.

Alg. 1 maintains finite sets Aw and Bw of words representing solutions of $A[\bar{s}]$ and $B[\bar{s}]$, respectively. In line 2, a candidate interpolant of the form $s_1|s_2|\cdots|s_n \in \mathcal{R}$ is constructed, in such a way that $\mathcal{L}(\mathcal{R})$ is a superset of Aw but disjoint from Bw. The concrete construction of candidate interpolants of size $\leq L$ can be implemented in a number of ways, for instance via an encoding as a SAT or SMT problem (as done in our implementation), or with the help of learning algorithms like L^* [1]. It is then checked whether $s_1|s_2|\cdots|s_n \in \mathcal{R}$ satisfies the properties of an interpolant (lines 3, 5), which can be done using the string solver developed in this paper. If any of the properties is violated, the constructed satisfying assignment η gives rise to a further word to be included in Aw or Bw.

Lemma 2 (Correctness). *Suppose bound L is chosen such that it is only satisfied by finitely many formulae $s_1|s_2|\cdots|s_n \in \mathcal{R}$. Then Alg. 1 terminates and either returns a correct interpolant $s_1|s_2|\cdots|s_n \in \mathcal{R}$, or reports **Inseparable**.*

By iteratively increasing bound L, eventually a regular interpolant for any (unsatisfiable) conjunction $A[\bar{s}] \wedge B[\bar{s}]$ can be found, provided that such an interpolant exists at all. This scheme of bounded interpolation is suitable for integration in the complete model checking algorithm given in [17]: since only finitely many predicates can be inferred for every value L, divergence of model checking is impossible for any fixed L. By globally increasing L in an iterative manner, eventually every predicate that can be expressed in the form $s_1|s_2|\cdots|s_n \in \mathcal{R}$ will be found.

7 Implementation

We have implemented our algorithm in a tool called NORN[5]. The tool takes as input a formula in the logic described in Section 3, and returns either *Sat* together with a witness of satisfiability (i.e. concrete string values for all variables), or *Unsat*. NORN first converts the given formula to DNF, after which each disjunct goes through the following steps:

1. Recursively split equalities, backtracking if necessary, until no equality constraints are left.
2. Recursively split membership constraints, again backtracking if necessary, and compute the langauge of each variable. From the langauge, we extract length constraints which we add to the formula.
3. Solve the remaining length constraints using PRINCESS [3].

We will now explain the second step in more detail. Assume that we have a membership constraint $tr \in A$, where A is an automaton (NORN makes use of DK.BRICS.AUTOMATON [23] for all automata operations). We can remove a sequence of trailing constants $a_1 a_2 \cdots a_k$ in $tr = tr' \cdot a_1 a_2 \cdots a_n$ by replacing the constraint with $tr' \in rev(\delta_{a_k \cdots a_2 a_1}(rev(A)))$, where $\delta_s(A)$ denotes the derivative of A w.r.t. the string s, and $rev(A)$ denotes the reverse of A. We now have a membership constraint $s_1 \cdots s_n \in A'$ where the term consists of a number of segments s_i, each of the form $a_1 \cdots a_{n_i} X_i$, i.e. a number of constants followed by a variable. The procedure keeps, at each step, a mapping m that maps each variable to an automaton representing the language it admits. For the constraint to be satisfiable, the constraints $s_1 \in A'_1$ and $s_2 \cdots s_n \in A'_2$ must be satisfiable for some pair (A_1, A_2) in the splitting of A'. This means that we can update our mapping by $m(X_i) = m(X_i) \cap \delta_{a_1 \cdots a_{n_i}}(A_1)$ and recurse on $s_2 \cdots s_n \in A'_2$. If at any point any automaton in the mapping becomes empty, the membership constraint is unsatisfiable, and we backtrack.

If, in the third step, PRINCESS tells that the given formula is satisfiable, it gives concrete lenghts for all variables. By restricting each variable to the solution given by PRINCESS and reversing the substitutions performed in step 1, we can compute witnesses for the variables in the original formula.

NORN can be used both as a library and as a command line tool. In addition to the logic in Section 3, NORN supports character ranges (e.g. $[a - c]$) and the wildcard character (.) in regular expressions. It also supports the divisibility predicate $x\, div\, y$, which says that x divides y. This translates to the arithmetic constraint $x = y * n$, where n is a free variable. However, the implementation does not currently support disequalities. Instead, they are handled by the model checker in the examples in which they arise.

Model Checking. We have integrated NORN into the predicate abstraction-based model checker Eldarica [15], on the basis of the algorithm and interpolation

[5] Available at `http://user.it.uu.se/~jarst116/norn/`. MD5 (string-mc.tar.gz) = 9ac43b07803fb69128a7009433d8d1e7

Program	**Property**	**Verification**
$a^n b^n$ (Fig. 1)	$s \notin (a+b)^* \cdot ba \cdot (a+b)^* \wedge \exists k.\ 2k = \lvert s \rvert$	Autom. (24.1s)
StringReplace	pre: $s \in (a+b+c)^*$; post: $s \in (a+c)^*$	Autom. (42.6s)
ChunkSplit	pre: $s \in (a+b)^*$; post: $s \in (a+b+c)^*$	Autom. (29.9s)
Levenshtein	$\mathit{dist} \leq \lvert s \rvert + \lvert t \rvert$	Autom. (9.4s)
HammingDistance	$\mathit{dist} = \lvert v \rvert$ if $u \in 0^*, v \in 1^*$	Autom. (2m29s)

Table 1. Verification results for a set of string-processing programs. Experiments were done on an Intel Core 2 Duo T9300 with 2.5GHz.

procedure from Section 6. We use the regular interpolation procedure from Section 6.2 in combination with an ordinary interpolation procedure for Presburger arithmetic to infer predicates about word length. Table 1 gives an overview of preliminary results obtained when analysing a set of hand-written string-processing programs. Although the programs are quite small, the presence of string operations makes them intricate to analyse using automated model checking techniques; most of the programs require invariants in form of regular expressions for verification to succeed. Our implementation is currently quite slow (it has not been optimised for performance yet), but able to verify all of the programs fully automatically.

We tried to analyse the programs also using the state-of-the-art model checkers UFO, SatAbs, and CPAchecker, but were not able to obtain positive results.

8 Conclusions and Future Work

In contrast to much of the existing work that has focused on the detection of bugs or the synthesis of attacks for string-manipulating programs; the work presented in this paper primarily targets verification of *functional correctness.* To achieve this goal, we have made several key contributions. First, we have presented a decision procedure for a rich logic of strings. Although the problem in its generality remains open, we are able to identify an expressive fragment for which our procedure is both sound and complete. We are not aware of any decision procedure with a similar expressive power. Second, we leverage on the fact that our logic is able to reason both about length properties and regular expressions in order to capture and manipulate predicates sufficient for a wide range of verification applications. Future works include experimenting with better integrations of the different theories, exploring different Craig interpolation techniques, and exploring the applicability of our framework to more general classes of string processing applications.

References

1. Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

2. Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A decade of software model checking with slam. *Commun. ACM*, 54(7):68–76, 2011.
3. Angelo Brillout, Daniel Kroening, Philipp Rümmer, and Thomas Wahl. An interpolating sequent calculus for quantifier-free Presburger arithmetic. *Journal of Automated Reasoning*, 47:341–367, 2011.
4. J. R. Büchi and S. Senger. Definability in the existential theory of concatenation and undecidable extensions of this theory. *Z. Math. Logik Grundlagen Math.*, 34(4), 1988.
5. Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT solver. In *TACAS*, pages 93–107, 2013.
6. William Craig. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *The Journal of Symbolic Logic*, 22(3), 1957.
7. Leonardo De Moura and Nikolaj Bjørner. 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/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.
8. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.
9. Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.
10. Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 519–531, Berlin, Heidelberg, 2007. Springer-Verlag.
11. Vijay Ganesh, Mia Minnes, Armando Solar-Lezama, and Martin Rinard. Word equations with length constraints: Whats decidable? In Armin Biere, Amir Nahir, and Tanja Vos, editors, *Hardware and Software: Verification and Testing*, volume 7857 of *Lecture Notes in Computer Science*, pages 209–226. Springer Berlin Heidelberg, 2013.
12. Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.
13. Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416, 2012.
14. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *31st POPL*, 2004.
15. Hossein Hojjat, Filip Konecný, Florent Garnier, Radu Iosif, Viktor Kuncak, and Philipp Rümmer. A verification toolkit for numerical transition systems - tool paper. In *FM*, pages 247–251, 2012.
16. Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4), 2009.
17. Ranjit Jhala and Kenneth L. McMillan. A practical and complete approach to predicate refinement. In *TACAS*, pages 459–473, 2006.

18. A. Kieżun, V. Ganesh, S. Artzi, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Transactions on Software Engineering and Methodology*, 21(4), 2012.
19. V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady.*, 10(8):707–710, February 1966.
20. G.S. Makanin. The problem of solvability of equations in a free semigroup. *Mathematics of the USSR-Sbornik*, 32(2):129–198, 1977.
21. Kenneth L. McMillan. Lazy abstraction with interpolants. In *CAV*, 2006.
22. Mario Méndez-Lojo, Jorge A. Navas, and Manuel V. Hermenegildo. A flexible, (c)lp-based approach to the analysis of object-oriented programs. In *LOPSTR*, 2007.
23. Anders Møller. dk.brics.automaton – finite-state automata and regular expressions for Java, 2010. `http://www.brics.dk/automaton/`.
24. Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Classifying and solving horn clauses for verification. In *VSTTE*, pages 1–21, 2013.
25. Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A Symbolic Execution Framework for JavaScript. In *IEEE Symposium on Security and Privacy*, pages 513–528. IEEE Computer Society, 2010.
26. Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. FLAX: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *NDSS*. The Internet Society, 2010.
27. Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Stranger: An automata-based string analysis tool for PHP. In *TACAS*, volume 6015 of *LNCS*, pages 154–157. Springer, 2010.
28. Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 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, pages 114–124, New York, NY, USA, 2013. ACM.

A Proof of Termination of the Decision Procedure

In this section, we will call equalities and disequalities equations and we will use $\approx$ to denote one of $=, \neq$. For a term tr, we will denote by tr_i the variable or letter on its ith position. For components of labels of edges of graphs, we will $\_$ as a placeholder to denote an existentially quantified component which is not important in the current context. We say that a variable u reaches v in a graph G_ϕ if $u = v$ or there is a path from u to v in G_ϕ.

A.1 Proof that the Rules Preserve Acyclic Form.

Lemma 3. *Let ϕ be in the acyclic form and let $u_1, \ldots, u_n$ be distinct fresh variables. Then $\phi[u/u_1 \cdots u_n]$ is in the acyclic form.*

Proof. By the definition of substitution and since $u_1 \cdots u_n$ are fresh, there is an edge between u and a variable v in G_ϕ labeled by $e(\_)$ iff for each $1 \leq i \leq n$, there is (exactly one) edge between u_i and v in $G_{\phi[u/u_1 \cdots u_n]}$ labeled by $e(\_)$. That means that (1) two variables x and y different from u and v were connected in G_ϕ if and only if they are connected in $G_{\phi[u/u_1 \cdots u_n]}$, (2) x was connected with v iff it is connected with all u_is, and (3) u was connected with u iff all u_is are connected with all u_is in $G_{\phi[u/u_1 \cdots u_n]}$. Any new cycle $\phi[u/u_1 \cdots u_n]$ must involve some new edges, hence some u_i, but this would mean that there was a cycle on u even in G_ϕ, which is a contradiction. $\square$

Lemma 4. *Let $\phi : u = tr \wedge \psi$ be in the acyclic form. Then $\psi[u/tr]$ is in the acyclic form.*

Proof. Notice first that tr nor contains u neither has two occurrences of a variable since $u = tr \wedge \psi$ is in the acyclic form. A loop can be introduced if $tr = tr' \cdot v \cdot tr''$ (notice that $u \neq v$) and there is $e : tr_1 \cdot u \cdot tr_2 \approx tr_3 \cdot x \cdot tr_4$ such that one of the two cases holds:

(1) x reaches v in G_ϕ. But because u is connected with v due to $u = tr$ and also connected with x due to e in G_ϕ, there is a loop in G_ϕ, which is a contradiction.

(2) tr_1 or tr_2 contain v. But because both u and v are connected with x by edges with $e(\_)$ and u is connected with v by an edge with $u = tr(\_)$, there was a loop in G_ϕ, which is a contradiction. $\square$

Lemma 5. *Application of rules of Section 4 preserves acyclic form.*

Proof.

Diseq-Split: We will show the claim for the case when the rule creates a formula defined by $\text{Split}_{\text{Diseq-Split}}$. The other two cases are analogical. The disequality $e : tr \neq tr'$ is transformed into $tr = u \cdot c \cdot v \wedge tr = u \cdot c' \cdot v'$ where u, v, v' are fresh. The difference between the graph of the original formula and the graph of the formula after the application of the rule is the following. All edges that do not have the label e stay. All edges that do have the label e are removed. Added are edges between x and u and between x and v for every x in tr, and edges between

y and u and between y and v' for every y in tr'. A loop can be introduced only due to the added edges. However, since variables u, v, v' are fresh, they do not appear in any other edges than in the added ones. The added edges cannot form a loop only by themselves. Hence they mus form a new connection between some variables z, z' different from u, v, v'. However, notice that the new edges may form at most one path from z to z', (it is leading via u). Otherwise one of the terms tr, tr' would contain a repetition of some variable and hence there would be a loop before the application of the rule. Since this connection of z, z' by addition of the new edges closed a loop, there is a path p between z, z' which does not involve label $e(\_)$, and this path was present even before application of the rule. But the addition of the new edges may interconnect only variables which were connected also originally by a label $e(\_)$. Hence the original graph contained a loop composed from p and an edge connecting z and z' labeled by $e(\_)$, which is a contradiction.

A-Eq-Var, Eq-Word: We will show the claim for the A-Eq-Var. The proof for the Eq-Word is analogical but simpler since there is less substitution and the problem is instead moved to regular constraints which do not concern acyclic form. We concentrate only on the case Split$'$ which is more complex than the non-primed case since it has more substitution.

Notice that since the original formula is in the acyclic form, we have that $u \neq v$, otherwise there would be a loop on u. The processing of the equality $e : u \cdot tr_1 = tr_2$ by the rule may be seen as three consecutive steps which all preserve acyclic form:

1. substitution of v by $v_1 \cdot v_2$,
2. removal of u from the left and removal of $tr_3 \cdot v_1$ from the right-hand side of e and subsequent addition of the equality $u = tr_3 \cdot v_1$,
3. the substitution of u by tr and removal of $u = tr$.

The first step preserves acyclic form by Lemma 3. The third step preserves acyclic form by Lemma 4. The second step needs more attention:

Let the removal of the prefixes of the sides of e in the first phase of the second step transform e to e'. This obviously preserves acyclic form since the new graph can only have less edges. The subsequent addition of $u = tr$ can add one edge between u and z labeled by $u = tr(u : 1, z : j)$ only if there was an edge between u and z labeled by $e(u : 1, z : j)$ that was removed by the removal of the prefixes. Hence the graph of the formula has the same or smaller number of edges between each pair of nodes then before the step. Since before there was no loop, there is no loop after the step.

Eq preserves acyclic form since it only removes an equality.

The other rules preserve acyclic form since they do not alter equalities and disequalities. □

A.2 Proof that the Decision Procedure Terminates

To prove termination of the decision procedure of Section 4, we need to take special care of the rules that process equalities. We will define a measure on conjuctions of equalities in the acyclic form which decreases with every application of the rule A-EQ-VAR, and which is 0 only if there are no equalities left.

Let E be a conjunction of equalities in the acyclic form without any constants. Let ρ be a map that assigns to every variable u a term $u_1 \cdots u_n$. We write $\rho^{-1}(u_i)$ to denote u. For a term tr and formula ϕ, we denote by $tr[\rho]$ and $\phi[\rho]$ the term and the formula, respectivelly, where every variable u was substitued by $\rho(u)$. ρ is a *solution* of E if for every equality $tr = tr'$ of E, it holds that $tr[\rho] = tr'[\rho]$, Define $\sim$ as the relation between variables that appear in the terms in the image of ρ such that $u \sim v$ iff there is an equation $tr = tr'$ of E and i such that $tr[\rho]_i = u$ and $tr'[\rho]_i = v$. We say that ρ *cuts* a variable u at the position i by a variable v if $\rho[u] = u_1 \cdots u_n, \rho(v) = v_1 \cdots v_m$ and for some $1 \leq i < n$, $u_i \sim^+ v_m$. Sim $\sim^+$ here denotes the transitive closure of $\sim$. We say that ρ *directly cuts* u at i by v in the same way as previously cut, except that we replace $\sim^+$ by $\sim$ (direct cut is a cut caused by a single equality). A solution ρ is *minimal* if for every u with $\rho(u) = u_1 \cdots u_n$, u is cut in ρ at every position $1 \leq i < n$. We define the size of ρ, denoted $|E|_\rho$, as the sum of lengths of all terms in $E[\rho]$. The size of E, denoted $|E|$, is then defined as the minimal size of its solution. Notice that if a solution ρ is not minimal due to that u is not cut at ith position, than it can be transformed into a solution ρ' by removing all variables z with $u_i \sim z$ from the image of ρ. We call ρ' a *simplification* of ρ. Notice that simplification yields a solution of a smaller size, and that by a chain of simplification, we can always arrive to a minimal solution from a non-minimal solution.

Lemma 6. *The set of minimal solution of a conjunction of equations E in acyclic form is finite (up to renaming of variables) and there is always at least one minimal solution.*

Proof (Sketch).
We first discuss computation of minimal solutions of a single equation $e = tr = tr'$ in the acyclic form. This is relatively straightforward since due to the acyclic form, no variable appears twice in e (neither on both sides nor twice within one side). Observe that minimal solutions of a single equality e in the acyclic form may be identified by sets of pairs of variables from e where a pair (u, v) is present in the set S_ρ associated with a solution ρ iff ρ directly cuts u by v. Hence, different minimal solutions of e have different sets of such pairs. Minimal solutions can thus be enumerated by enumerating possible sets S_ρ (and there is finitely many of them).

Any possible minimal solution of a conjunction E can be computed as follows. Fix an ordering $e_1, \ldots, e_n$ of the equations in E. Process the equations one by one in the ordering as follows: Let $e : tr = tr'$ be the equation processed. Compute a minimal solution ρ for e. Replace every equation of E by e_i by $e_i[\rho]$. Move to the next equation in the ordering (to its replacement), or, if e was the last one, return ρ as a possible minimal solution. The procedure terminates since there is a finitely many equations and computation of a minimal solution of an equation in acyclic form terminates. There is only a finitely many possible results of the computation since (1) there is only finitely many orderings of E, (2) there is only a finitely many minimal solutions of a single equation in the acyclic form, and, (3) after the substitution, all $e_i[\rho]$s are still in the acyclic form by Lemma 4, hence every iteration is performed with e in the acyclic form.

There is always at least one solution which assigns empty words to all variables.

□

Lemma 7. *It is possible to define a size function for conjunctive formula such that the application of a rule* A-EQ-VAR *will leads to a set of conjunctive formulas with a strictly decreasing size value.*

Proof. We will show that the claim holds for A-EQ-VAR. The proof the is analogical. The rule may be seen as working in three steps. The first step is the substitution of u by tr where $tr = tr_3 \cdot v_1$ or $tr = tr_3$. The second step is the substitution of v by $v_1 \cdot v_2$ (optional). The third step is the removal of u and removal of the tr from the right-hand side.

The first step does not increase the size of E. To see this, let the first step transform E into E_1. Let E' and E'_1 be the sets of equations with constants removed and let tr' be tr with constants removed. Then E'_1 is obtained from E' by substituting u by tr'. Hence for every solution ρ_1 of E'_1, there is a solution ρ of E' such that if $tr'[\rho_1] = v_1 \cdots v_n$, then $\rho(u) = u_1 \cdots u_n$ and such that on variables different from u, ρ is the same as ρ_1. ρ has the same size as ρ_1. ρ is either minimal or some of its simplifications (with even smaller size) is minimal. Hence for every minimal solution of E'_1, there is a minimal solution of E' which has a larger or the same size, and so $|E_1| \leq |E_2|$.

The second step does not increase the size of E_1. To see this, let the second step transform E_1 into E_2. Notice that for every minimal solution of E'_2 (E_2 with constants removed) with $\rho_2(v_1) = u^1_1, \ldots, u^1_n$ and $\rho_1(v_2) = u^2_1, \ldots, u^2_m$, there is a solution ρ_1 of E'_1 which is the same as ρ_2 up to that instead of defining the value for v_1 and v_2, it has $\rho_1(v) = u^1_1, \ldots, u^1_n, u^2_1, \ldots, u^2_m$. It can be seen that $|E'_1|_{\rho_1} = |E'_2|_{\rho_2}$. Either ρ_1 is minimal or there is a minimal solution for E'_1 of a smaller size. Hence for every minimal solution of E'_2, there is a minimal solution of E'_1 which has a larger or the same size, and so $|E_2| \leq |E_1|$.

Last, we argue that the third step, which shortens an equality and transforms the set of equalities E_2 into E_3, decreases the size. Let E'_3 be E_3 with constants removed. Clearly any solution ρ of E'_2 is a solution of E'_3. All terms appearing in E'_3 are the same or shorter than corresponding terms appearing in E'_2, hence the size of ρ in E'_3 is smaller than its size in E'_2. Every minimal solution ρ of E'_3 is a minimal solution of E_2 or it is a simplification (w.r.t E'_3) of some minimal solution ρ' of E'_2. In the first case, it must hold that $|E'_3|_\rho < |E'_2|_\rho$, and in te second case that $|E'_3|_\rho < |E'_2|_{\rho'}$. Hence for every minimal solution of E'_3, there is a minimal solution of E'_2 which has a larger size, and so $|E_3| < |E_2|$.

Since the first two steps do not increase the size and the last step decreases the size, application of the rule A-EQ-VAR decreases the size of the conjunction of equalities. Similar arguments can be used in the case A-EQ-VAR. □

Theorem 1. *The decision procedure of Section 4 terminates for formulas in acyclic forms.*

Proof. Let ϕ be a formula in acyclic form. We assume w.l.o.g that it is in the disjunctive normal form (i.e., $\phi = \phi_1' \cup \cdots \phi_n'$ where ϕ_1' is a conjunctions of literals) . It is easy to see that the labels of the proof tree rooted at ϕ can be transformed in the disjunctive normal form.

We will show termination by contradiction but first we recall that the following rules can be applied only finite number of times:

– Each application of Diseq-Split removes one disequality and other rule adds a disequality. Thus in any branch of the proof tree the number of times that Diseq-Split can be applied is bounded by the initial number of disequality.
– Not-Eq can only be applied once in any branch of the proof tree since it closes the branch.
– Eq can be applied only finite number of times (bounded by the number of equalities and disequalities) in the case of acyclic formula. In fact, all the rules that can be applied to an acyclic formula do not increase the number of equalities and disequalities.
– C-Eq-Var will never be applied due to the acyclicity.
– Eq-Word removes at least one occurrence of a constant from the set of equalities and disequalities and there is no rule which would increase the number of occurrences of constants in the set. So the number of application of this rules is bounded by the number of occurrences of constants in ϕ.
– Reg-Neg decreases the number of negations in the formula and no other rule increases it.
– A-Eq-Var decreases the size of the equalities by Lemma 7. When the size is 0, there are no equalities left (observe that the rule A-Eq-Var can be always applied if there is an equality equation). The only rules that may increase the size are A-Eq-Word and Diseq-Split. But we showed that they can be used only finitely many times. Therefore, from some point on, the size of equalities is only decreasing with every applicatio of A-Eq-Var. Hence it can be used only finitely many times.
– Reg-Split splits a term within a membership predicate. This decreases the sum of concatenations in all terms in membership predicates. The rules that can increase the sum are the rules for removing equalities, but these were shown to be used only finitely many times.
– Memb, Not-Memb, and Reg-Lenght decrease the number of membership predicates. Moreover, they are bounded by the number of application of the other rules that were shown to be used only finitely many times.
– Term-Length decreases the number of concatenation in the arithmetic formula, which can increase by a use of other rules, hence it is bounded by the application of the other rules. The other rules were shown to be used only finitely many times.

We have shown that every rule can be used only finitely many times within a branch of the proof tree. Hence every branch is finite. The tree has a finite degree, and therefore, by König's lemma, it cannot be infinite (since it deos not contain an infinite branch).

□