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 length of words, and on (iii) the regular languages to which words belong. Decidability of this general logic is still open. Our procedure is sound for the general logic, and a decision procedure for a particularly rich fragment that restricts the form in which word equations are written. In contrast to many existing procedures, our method does not make assumptions about the maximum length of words. We have developed a prototypical implementation of our decision procedure, and integrated it into 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 [15, 8]. Mature model checking tools are already used in industrial applications [2]. One main reason for this success are recent developments in SMT technology [5, 7, 3], which enable efficient reasoning about symbolic representations of different data types in programs. This dependence encompasses, however, 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 the length of words, 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 length are either unsound, not expressive enough, or lack the ability to provide counterexamples. Many solvers [9, 23, 24] are unsound since they assume an a priori fixed upper bound on the length of the possible words. Others [9, 17, 26] are not expressive enough as they do not handle word equations, length constraints, or membership predicates. Such solvers are mostly aimed at performing symbolic executions, i.e., establishing feasibility of paths in a program. The solver in [25] performs sound over-approximation, but without supplying counterexamples 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 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 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 analyze programs that are currently beyond the reach of state-of-the-art software model checkers.

Related Work. The pioneering work by Makanin [18] 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 $|u| = |v|$. Our logic 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 sub-logic for which we guarantee the termination of our procedure.

In a work close to ours, the authors in [10] show decidability of a logic that is strictly weaker than the one for which we guarantee termination. For instance, in [10], 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 expressiveness of the logic. In [26], 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 [10, 26] 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 `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 string variables is necessarily unsound and will fail to establish correctness.

Moreover, establishing correctness requires the ability to express and to reason about predicates such as those mentioned in the comments of the code in

```
// Pre = (true)
String s= '';
// P1 = (s ∈ ϵ)
while(*){
    // P2 = (s = u · v ∧ u ∈ a* ∧ v ∈ b* ∧ |u| = |v|)
    s= 'a' + s + 'b';
}
// P3 = P2
assert(!s.contains('ba') && (s.length() % 2) == 0);
// Post = P3
```

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.

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

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

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 guarantee 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, regular constraints (short for membership constraints in regular languages) and length and arithmetic inequalities. 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:

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

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 occurrence of u_i by term tr_i (resp. each occurrence of $|u_i|$ by expression e_i). Such a substitution is said to be well-defined if no variable u_i (resp. $|u_i|$) appears in any tr_i (resp. e_i).

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 Inference Rules

In this section, we describe our set of inference rules for checking the 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:

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

In this inference rule, Name is the name of the rule, *cond* is a side condition 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. (We omit the side condition *cond* from Name when it is *tt*.) 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.

Lemma 1. *The inference rules of this section are sound and locally complete.*

4.1 Removing Disequalities

We use rules Not-Eq and Diseq-Split in order to eliminate disequalities. In rule Not-Eq, 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 $\textsc{Split}''_{\textsc{Diseq-Split}} = \{tr = u \wedge tr' = u \cdot c \cdot v' \wedge \phi \mid c \in \Sigma\}$.

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

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

4.2 Simplifying Equalities

We introduce rules Eq, 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 $tr = tr$.

Rule Eq-Var eliminates variable u from the equality $u \cdot tr_1 = tr_2 \wedge \phi$. Let w_u be some word denoted by u. For the equality to hold, w_u must be a prefix of the word denoted by tr_2. There are two cases. The first case, represented by $\textsc{Split}_{\textsc{Eq-Var}}$ in Eq-Var, captures situations where w_u coincides with a word denoted by a prefix tr_3 of tr_2. The second case, represented by $\textsc{Split}'_{\textsc{Eq-Var}}$, captures situations where w_u does not coincide with a word denoted by a prefix of 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.

$$\textsc{Eq-Var} : \frac{\textsc{Split}_{\textsc{Eq-Var}} \cup \textsc{Split}'_{\textsc{Eq-Var}}}{u \cdot tr_1 = tr_2 \wedge \phi}$$

The set $\textsc{Split}_{\text{Eq-Var}}$ captures the first case, when w_u coincides with a word denoted by a prefix tr_3 of tr_2. The premises for this case are partitioned into two sets, namely $\textsc{Split}_{\text{Eq-Var-1}}$ and $\textsc{Split}_{\text{Eq-Var-2}}$:

$$\textsc{Split}_{\text{Eq-Var-1}} = \left\{ \begin{array}{l} (tr_1 = tr_4 \wedge \phi)[u/tr_3] \mid \\ tr_2 = tr_3 \cdot tr_4 \text{ and } u \text{ does not syntactically appear in } tr_3 \end{array} \right\}$$

$$\textsc{Split}_{\text{Eq-Var-2}} = \left\{ \begin{array}{l} tr_1 = tr_4 \wedge tr_5 \cdot tr_6 \in \epsilon \wedge \phi \mid \\ tr_2 = tr_3 \cdot tr_4 \text{ and } tr_3 = tr_5 \cdot u \cdot tr_6 \end{array} \right\}$$

Variable u is eliminated from the premises contained in the set $\textsc{Split}_{\text{Eq-Var-1}}$. The second set $\textsc{Split}_{\text{Eq-Var-2}}$ captures cases where u does syntactically appear in tr_3. Variable u might still appear in some of the premises of $\textsc{Split}_{\text{Eq-Var-2}}$.

The set $\textsc{Split}'_{\text{Eq-Var}}$ in $\textsc{Eq-Var}$ captures the second case, namely when w_u does not coincide with a word denoted by a prefix of tr_2, written $tr_3 \cdot v \cdot tr_4$ for some variable v. The premises in $\textsc{Split}'_{\text{Eq-Var}}$ are partitioned into two sets, namely $\textsc{Split}'_{\text{Eq-Var-1}}$ and $\textsc{Split}'_{\text{Eq-Var-2}}$:

$$\textsc{Split}'_{\text{Eq-Var-1}} = \left\{ \begin{array}{l} \big((tr_1 = v_2 \cdot tr_4 \wedge \phi)[u/tr_3 \cdot v_1]\big)[v/v_1 \cdot v_2] \mid \\ tr_2 = tr_3 \cdot v \cdot tr_4 \text{ and } u \text{ appears neither in } tr_3 \text{ nor in } v \end{array} \right\}$$

$$\textsc{Split}'_{\text{Eq-Var-2}} = \left\{ \begin{array}{l} \big(tr_1 = u_2 \cdot tr_4 \wedge u_1 \cdot u_2 = tr_3 \cdot u_1 \wedge \phi\big)[u/tr_3 \cdot u_1] \mid \\ tr_2 = tr_3 \cdot u \cdot tr_4 \text{ and } u \text{ does not appear in } tr_3 \end{array} \right\}$$

The premises in $\textsc{Split}'_{\text{Eq-Var-1}}$ mention neither u nor v. The set $\textsc{Split}'_{\text{Eq-Var-2}}$ captures cases where u in the left-hand side overlaps with its occurrence on the right-hand side. Cases where u appears in tr_3 are captured in $\textsc{Split}_{\text{Eq-Var}}$.

Rule $\textsc{Eq-Word}$ eliminates the word w from the equality $w \cdot tr_1 = tr_2 \wedge \phi$:

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

Again, we define two sets representing the premises of the rule:

$$\textsc{Split}_{\text{Eq-Word}} = \big\{ tr_3 \in w \wedge tr_4 = tr_1 \wedge \phi \mid tr_2 = tr_3 \cdot tr_4 \big\}$$
$$\textsc{Split}'_{\text{Eq-Word}} = \big\{ \big(tr_3 \cdot v_1 \in w \wedge v_2 \cdot tr_4 = tr_1 \wedge \phi\big)[v/v_1 \cdot v_2] \mid tr_2 = tr_3 \cdot v \cdot tr_4 \big\}$$

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

4.3 Removing Membership Predicates

We use rules $\textsc{Reg-Neg}$, $\textsc{Memb}$, $\textsc{Not-Memb}$, $\textsc{Reg-Split}$ and $\textsc{Reg-Len}$ to simplify and eliminate membership predicates. We describe them below.

Rule $\textsc{Reg-Neg}$ 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 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 $\bigvee_{i=1}^{n} \left(tr \in \mathcal{R}_i \wedge tr' \in \mathcal{R}'_i\right)$ 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-LEN 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 predicate. The rule REG-LEN 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 singleton, its form will be either a lasso or a simple path.) After this step, there will be no membership predicates involving u.

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

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 $|tr \cdot tr'|$.

$$\textsc{Term-Leng}: \frac{\phi[|tr \cdot tr'|/|tr| + |tr'|]}{\phi}\ |tr \cdot tr'| \text{ appears in } \phi$$

We can also add rules to systematically add the length predicate $|tr| = |tr'|$ each time an equality $tr = tr'$ appears in a formula; however, such rules are not necessary for the completeness of our procedure, as shown in the next section.

5 Completeness of the Procedure

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

Non-termination may be caused by an infinite chain of applications of rule EQ-VAR of Section 4.2 for removing equalities. Consider for instance the equality $u \cdot v = v \cdot u$. One of the cases generated within the disjunct $\text{SPLIT}'_{\text{EQ-VAR-1}}$ of 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. We also need to prevent the repetition of a variable inside one side of an equality. This is needed in cases like $u \cdot u = v \cdot v$ where $\text{SPLIT}'_{\text{EQ-VAR-1}}$ includes $v_1 = v_2 \cdot v_1 \cdot v_2$, with a variable v_1 on both sides of the equality, which is the situation which we wanted to prevent at the first place. These restrictions must hold initially and must be preserved by applications of any of the rules presented in Sections 4. Attention must be given to rules that modify equalities. Rules such as 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 several times in a (dis)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. We respectively state preservation of these restriction and termination of the procedure of Section 4 in theorems 1 and 2 at the end of this Section. First, we need some definitions.

Linear formulae. A formula in $\mathcal{E}_{e,r,l}$ is said to be *linear* if it contains no (dis)equality where a variable appears more than once.

Given a conjunction ϕ in $\mathcal{E}_{e,r,l}$ involving m (dis)equalities, we can build a *dependency graph* $G_\phi = (N, E, \texttt{label}, \texttt{map})$ in the following way. We order the (dis)equalities from e_1 to e_m, where each e_j is of the form $\texttt{lhs}(j) \approx \texttt{rhs}(j)$ for $j : 1 \leq j \leq m$ and $\approx \in \{=, \neq\}$. For each $j : 1 \leq j \leq m$, a node n_{2j-1} is used to refer to the left-hand side of the j^{th} (dis)equality, and n_{2j} to its right-hand side. For example, two different nodes are used even in the case of the simple equality $u = u$, one to refer to the left-hand side, and the other to refer the right-hand side. N is then the set of $2 \times m$ nodes $\{n_i | i : 1 \leq i \leq 2 \times m\}$. The mapping $\texttt{label}$ associates the term $\texttt{lhs}(j)$ (resp. $\texttt{rhs}(j)$) to each node n_{2j-1} (resp. n_{2j}) for $j : 1 \leq j \leq m$. $\texttt{label}$ is not necessarily a one to one mapping. The mapping $\texttt{map} : E \to \{\texttt{rel}, \texttt{var}\}$ labels edges as follows: $\texttt{map}(n, n') = \texttt{rel}$ for each $(n, n') = (n_{2j-1}, n_{2j})$ for each $j : 1 \leq j \leq m$, and $\texttt{map}(n, n') = \texttt{var}$ iff

$n \neq n'$, and $\texttt{label}(n)$ and $\texttt{label}(n')$ have some common variables. By construction, $\texttt{map}$ is defined to be total, i.e., E contains only edges that are labeled by $\texttt{map}$.

A *dependency cycle* in $G_\phi = (N, E, \texttt{label}, \texttt{map})$ is a cycle where successive edges have alternating labels. Formally, a dependency cycle is a sequence of distinct nodes $n_0, n_1, \ldots, n_k$ in N with $k \geq 1$ such that 1) for every $i : 0 \leq i \leq k$, $\texttt{map}(n_i, n_{i+1\%(k+1)})$ is defined, and 2) for each $i : 0 \leq i < k$, $\texttt{map}(n_i, n_{i+1}) \neq \texttt{map}(n_{i+1}, n_{i+2\%(k+1)})$.

Acyclic graph. A conjunction ϕ in $\mathcal{E}_{e,r,l}$ is said to be acyclic iff it is linear and its dependency graph does not contain any dependency cycle.

Theorem 1. *Application of rules of Section 4 preserves acyclicity.*

An *ordered procedure* is any procedure that applies the rules of Section 4 on a formula in $\mathcal{E}_{e,r,l}$ in the four following phases. In the first phase, all disequalities are eliminated using Diseq-Split and Not-Eq. In the second phase, the procedure eliminates one equality at a time by repeatedly applying Eq-Var, Eq-Word and Eq. In the third phase, membership predicates are eliminated by repeatedly applying Reg-Neg, Memb, Not-Memb, Reg-Split and Reg-Len. In the last phase, arithmetic predicates are solved using a standard decision procedure [3].

Theorem 2. *Ordered procedures terminate on acyclic formulae.*

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 [11, 13] or Impact-style algorithms [19]; 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 [16]. We represent programs in the framework of Horn clauses [20, 12], 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 $H \leftarrow C \land B_1 \land \cdots \land B_n$ 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.

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

Algorithms to construct solutions of Horn clauses with the help of *predicate abstraction* have been proposed for instance in [12]; 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 (of length *len*, starting at position *pos*) by v', resulting in the new overall string u':

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

- *search* for the first occurrence of a string, using either equations or regular expression constraints.

6.2 Constraint-Based Craig Interpolation

In order to synthesize 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

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* $Bw \cap \mathcal{L}(\mathcal{R}) = \emptyset$ **do**
3   **if** $A[\bar{s}] \wedge \neg(s_1|s_2|\cdots|s_n \in \mathcal{R})$ *is satisfiable with assignment* η **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* η **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***;

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 [22].

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 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 [16]: 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 language of each variable. From the language, 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 [21] 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 lengths 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

[5] Available at `http://user.it.uu.se/~jarst116/norn/`.

Program	Property	Time
$a^n b^n$ (Fig. 1)	$s \notin (a+b)^* \cdot ba \cdot (a+b)^* \wedge \exists k.\ 2k = \lvert s \rvert$	8.0s
StringReplace	pre: $s \in (a+b+c)^*$; post: $s \in (a+c)^*$	4.5s
ChunkSplit	pre: $s \in (a+b)^*$; post: $s \in (a+b+c)^*$	5.5s
Levenshtein	$dist \leq \lvert s \rvert + \lvert t \rvert$	5.3s
HammingDistance	$dist = \lvert v \rvert$ if $u \in 0^*, v \in 1^*$	27.1s

Table 1. Verification runtime for a set of string-processing programs. Experiments were done on an Intel Core i5 machine with 3.2GHz, running 64 bit Linux.

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.

Model Checking. We have integrated Norn into the predicate abstraction-based model checker Eldarica [14], on the basis of the algorithm and interpolation 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 analyzing a set of hand-written string-processing programs. Although the programs are quite small, the presence of string operations makes them intricate to analyze using automated model checking techniques; most of the programs require invariants in form of regular expressions for verification to succeed. Our implementation is able to verify all programs fully automatically within a few seconds; since performance has not been the main focus of our implementation work so far, further optimization will likely result in much reduced runtimes. To the best of our knowledge, all of the programs are beyond the scope of other state-of-the-art software model checkers.

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. 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.
9. 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.
10. 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.
11. Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.
12. Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416, 2012.
13. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *31st POPL*, 2004.
14. 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.
15. Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4), 2009.

16. Ranjit Jhala and Kenneth L. McMillan. A practical and complete approach to predicate refinement. In *TACAS*, pages 459–473, 2006.
17. 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.
18. G.S. Makanin. The problem of solvability of equations in a free semigroup. *Mathematics of the USSR-Sbornik*, 32(2):129–198, 1977.
19. Kenneth L. McMillan. Lazy abstraction with interpolants. In *CAV*, 2006.
20. 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.
21. Anders Møller. dk.brics.automaton – finite-state automata and regular expressions for Java, 2010. `http://www.brics.dk/automaton/`.
22. Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Classifying and solving horn clauses for verification. In *VSTTE*, pages 1–21, 2013.
23. 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.
24. 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.
25. 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.
26. 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.

9 Proof of Theorem 1

Proof. We only need to consider rules defined in Section 4.1 and Section 4.2 (since the other rules can not affect the acyclicity of the formula). Let ϕ be an acrylic formula of the form $\phi' \wedge (\bigwedge_{i=1}^{m} e_i)$ where $(tr_i \approx_i tr'_i)$ with $\approx_i \in \{=, \neq\}$ for all $i \in \{1, \ldots, m\}$, and ϕ' is a conjunction of length constraints and membership predicates. Let $G_\phi = (N_\phi, E_\phi, \mathtt{label}_\phi, \mathtt{map}_\phi)$ be the dependency graph associated to ϕ as defined 5.

Depending on the applied proof rule, we have four cases to study:

Diseq-Split: We will show the claim for the case when the rule creates a formula defined by $\textsc{Split}_{\textsc{Diseq-Split}}$. The other two cases are quite similar. Let us assume that the disequality $e_m : tr_m \neq tr'_m$ is transformed into $tr_m = u \cdot c \cdot v \wedge tr_{m+1} = u \cdot c' \cdot v'$ where u, v, v' are fresh. Then, the resulting formula ϕ_{res} is of the form $\phi' \wedge \bigwedge_{i=1}^{m+1} e'_i$ with $e_i = e'_i$ for all $i \in \{1, \ldots, m-1\}$, $e'_m = (tr_m = u \cdot c \cdot v)$ and $e'_{m+1} = (tr_{m+1} = u \cdot c' \cdot v')$. It is clear that ϕ_{res} is linear since ϕ is linear and u, v, v' are fresh variables.

Let $G_{\phi_{res}} = (N_{\phi_{res}}, E_{\phi_{res}}, \mathtt{label}_{\phi_{res}}, \mathtt{map}_{\phi_{res}})$ be the corresponding dependency graph for ϕ_{res}. We assume w.l.o.g that the set of nodes $N_{\phi_{res}} = \{n_i \,|\, 1 \leq i \leq 2m+2\}$ where the node n_{2i-1} is used to refer to the left-hand side of the i-th (dis)equality e'_i (or e_i) while the node n_{2i} is used to refer to the right-hand side of the i-th (dis)equality e'_i (or e_i) for all $i : 1 \leq i < m$. The node n_{2m-1} refers to left-hand side of the (dis)equality e'_m (or e_m) and n_{2m} left-hand side of the (dis)equality e'_{m+1} (or the right-hand side of the (dis)-equality e_m). The nodes n_{2m+1} and n_{2m+2} refer to the right-hand side of the (dis)-equality e'_m and e'_{m+1}. Then, it is easy to observe that $\mathtt{label}_\phi(n_i) = \mathtt{label}_{\phi_{res}}(n_i)$ for all $i : 1 \leq i \leq 2m$. The set of edges of $G_{\phi_{res}}$ is defined as the union of $E_\phi \setminus \{(n_{2m-1}, n_{2m})\}$ and $\{(n_{2m-1}, n_{2m+1}), (n_{2m}, n_{2m+2}), (n_{2m+1}, n_{2m+2})\}$. Furthermore, we have $\mathtt{map}_{\phi_{res}}(n, n') = \mathtt{map}_\phi(n, n')$ if $(n, n') \in E_\phi$, $\mathtt{map}_{\phi_{res}}(n_{2m-1}, n_{2m+1}) = \mathtt{rel}$, $\mathtt{map}_{\phi_{res}}(n_{2m}, n_{2m+2}) = \mathtt{rel}$, and $\mathtt{map}_{\phi_{res}}(n_{2m+1}, n_{2m+2}) = \mathtt{var}$.

Observe that the difference between the dependency graph of the original formula and the graph of the formula after the application of the rule is the following. We remove the $\mathtt{rel}$ -edge between n_{2m-1} and n_{2m} and we add three edges as follows: an $\mathtt{rel}$-edge between n_{2m-1} and n_{2m+1}, a $\mathtt{var}$-edge between n_{2m+1} and n_{2m+2}, and finally an $\mathtt{rel}$-edge between n_{2m} and n_{2m+2}.

The proof is done by contradiction. Let us assume that $G_{\phi_{res}}$ has a dependency cycle: There is sequence of distinct nodes $q_0, q_1, \ldots, q_k$ in $N_{\phi_{res}}$ with $k \geq 1$ such that 1) for every $i : 0 \leq i \leq k$, $\mathtt{map}(q_i, q_{i+1\%(k+1)})$ is defined, and 2) for each $i : 0 \leq i < k$, $\mathtt{map}(q_i, q_{i+1}) \neq \mathtt{map}(q_{i+1}, q_{i+2\%(k+1)})$.

Observe that there is an index $j : 0 \leq j \leq k$ such that $q_j \in \{n_{2m+1}, n_{2m+2}\}$, otherwise it is already a dependency cycle in G_ϕ (which is a contradiction). We assume that j is the smallest of such index. This implies that there is an index $\ell : j < \ell \leq k$ such that $q_\ell \in \{n_{2m+1}, n_{2m+2}\} \setminus \{q_j\}$ (since the node q_j is connected only by two edges and one of them is shared with q_ℓ). Moreover, it is easy to see that $k > 3$ since the nodes n_{2m-1} and n_{2m} should appear in the dependency cycles of G_ϕ.

There are only three cases to consider depending on the position of j and ℓ:

- If $1 \leq j < k-1$ then we have $\ell = j+1$ with $\ell < k$. This implies that the sequence of distinct nodes $q_0, q_1, \ldots, q_{j-1}, q_{j+2}, \ldots, q_k$ is a dependency cycle since the path $(q_{j-1}, q_j)(q_j, q_{j+1})(q_{j+1}, q_{j+1})$ in $G_{\phi_{res}}$ (with $\mathtt{map}(q_{j-1}, q_j) = \mathtt{rel}$, $\mathtt{map}(q_j, q_{j+1}) = \mathtt{var}$ and $\mathtt{map}(q_{j+1}, q_{j+2}) = \mathtt{rel}$) was replaced by the edge (q_{j-1}, q_{j+2}) with $\mathtt{map}(q_{j-1}, q_{j+2}) = \mathtt{rel}$ and so we respect the label alternation of the edges (which is a contradiction).
- If $j = k-1$ then we have $\ell = k$. This implies that the sequence of distinct nodes $q_0, q_1, \ldots, q_{k-2}$ is a dependency cycle since the path $(q_{k-2}, q_{k-1})(q_{k-1}, q_k)(q_k, q_0)$ in $G_{\phi_{res}}$ (with $\mathtt{map}(q_{k-2}, q_{k-1}) = \mathtt{rel}$, $\mathtt{map}(q_{k-1}, q_k) = \mathtt{var}$ and $\mathtt{map}(q_k, q_0) = \mathtt{rel}$) was replaced by the edge (q_{k-2}, q_0) with $\mathtt{map}(q_{k-2}, q_0) = \mathtt{rel}$ and so we respect the label alternation of the edges (which is a contradiction).

- If $j = 0$ and $\ell = 1$, then the sequence of distinct nodes $q_2, q_3, \ldots, q_k$ is a dependency cycle since the path $(q_k, q_0)(q_0, q_1)(q_1, q_2)$ in $G_{\phi_{res}}$ (with $\texttt{map}(q_k, q_0) = \texttt{rel}$, $\texttt{map}(q_0, q_1) = \texttt{var}$ and $\texttt{map}(q_1, q_2) = \texttt{rel}$) was replaced by the edge (q_k, q_2) with $\texttt{map}(q_k, q_2) = \texttt{rel}$ and so we respect the label alternation of the edges (which is a contradiction).
- If $j = 0$ and $\ell = k$, then the sequence of distinct nodes $q_1, q_2, \ldots, q_{k-1}$ is a dependency cycle since the path $(q_{k-1}, q_k)(q_k, q_0)(q_0, q_1)$ in $G_{\phi_{res}}$ (with $\texttt{map}(q_{k-1}, q_k) = \texttt{rel}$, $\texttt{map}(q_k, q_0) = \texttt{var}$ and $\texttt{map}(q_0, q_1) = \texttt{rel}$) was replaced by the edge (q_{k-1}, q_1) with $\texttt{map}(q_{k-1}, q_1) = \texttt{rel}$ and so we respect the label alternation of the edges (which is a contradiction).

SPLIT$_{\text{EQ-VAR}}$, EQ-WORD: We will show the claim for the SPLIT$_{\text{EQ-VAR}}$. The proof for the EQ-WORD is simpler since it does not introduce new nodes or edges in the dependency (modulo renaming).

Let first concentrate on the case SPLIT$_{\text{EQ-VAR-1}}$ (the other case SPLIT$_{\text{EQ-VAR-2}}$ can't be applied due to the acyclic formula since the variable u can't appear on both sides).

Let us assume that the equality $e_m = (tr_m = tr'_m)$, with $tr_m = u \cdot tr$ and $tr'_m = tr' \cdot tr''$, is transformed into $tr = tr''$. Then, the resulting formula ϕ_{res} is of the form $\phi'[u/tr'] \wedge (\bigwedge_{i=1}^{m} e'_i)$ with $e'_i = e_i[u/tr']$ for all $i \in \{1, \ldots, m-1\}$ and $e'_m = (tr = tr'')$.

The application of a such rule may be seen as two consecutive steps which preserve acyclic form:

1. removal of u and tr' from the equality $e_m = (tr_m = tr'_m)$. We obtain the formula ϕ_1 which is of the form $\phi' \wedge (\bigwedge_{i=1}^{m-1} e_i) \wedge (e'_m)$
2. substitution of u by tr' in ϕ_1 and which leads to the formula ϕ_{res}.

It is clear that ϕ_1 is linear and that the dependency graph $G_{\phi_1} = (N_{\phi_1}, E_{\phi_1}, \texttt{label}_{\phi_1}, \texttt{map}_{\phi_1})$ associated to ϕ_1 is free from dependency cycles since it consists of the graph G_ϕ where we have removed some var-edges. Hence ϕ_1 is an acyclic formula. We assume w.l.o.g that $N_{\phi_1} = N_\phi$, $\texttt{label}_{\phi_1}(n_j) = \texttt{label}_\phi(n_j)$ for all $j \in \{1, \ldots, 2m-2\}$ and n_{2m-1} (resp. n_{2m}) refers to the left-hand (right-hand) side of the equality e'_m.

Now after the application of step 1, we replace any occurrence of u by the term tr'. Let us assume that ϕ_{res} is not consistent. This means that there is a (dis)equality $e_i : tr_i[u/tr'] \approx_i tr'_i[u/tr']$ where a variable v appears at least twice. This means that v appears in tr' and appears in $tr_i \cdot tr'_i$. There are two cases: (1) u and v appears in tr_i (resp. tr'_i), then we can construct a dependency cycle in G_ϕ as follows $n_{2i-1}n_{2m-1}n_{2m}$ (reps. $n_{2i}n_{2m}n_{2m-1}$), which contradicts the acyclicity assumption of ϕ, or (2) u appears in tr_i and v in tr'_i (resp. v in tr_i and u in tr'_i) , then we can construct the following dependency graph $n_{2i-1}n_{2m-1}n_{2m}n_{2i}$ (resp. $n_{2i}n_{2m-1}n_{2m}n_{2i-1}$), which contradicts also the acyclicity assumption of ϕ. Hence ϕ_{res} is linear.

Let us show now that the dependency graph of ϕ_{res} does not contain any dependent cycle.

Let $G_{\phi_{res}} = (N_\phi, E_{\phi_{res}}, \texttt{label}_{\phi_{res}}, \texttt{map}_{\phi_{res}})$ be the corresponding dependency graphs for ϕ_{res}. The difference between the dependency graph of the formula ϕ_1 and the graph of the formula ϕ_{res} after the second step is that we add some `var` -edges to the graph G_{ϕ_1}.

The proof is done by contradiction. et us assume that $G_{\phi_{res}}$ has a dependency cycle: There is sequence of distinct nodes $q_0, q_1, \ldots, q_k$ in $N_{\phi_{res}}$ with $k \geq 1$ such that 1) for every $i : 0 \leq i \leq k$, $\mathtt{map}(q_i, q_{i+1\%(k+1)})$ is defined, and 2) for each $i : 0 \leq i < k$, $\mathtt{map}(q_i, q_{i+1}) \neq \mathtt{map}(q_{i+1}, q_{i+2\%(k+1)})$.

Observe that there is $j : 0 \leq j < k$ such that $\mathtt{map}_{\phi_{res}}(q_j, q_{j+1}) = \mathtt{var}$ is a newly added edge in G_ϕ with respect to G_{ϕ_1}, otherwise it is already a dependency cycle in G_{ϕ_1} (which is a contradiction).

Now we can through the dependency cycle $q_0, q_1, \ldots, q_k$ and replace any newly added edge (q_j, q_{j+1}) with $\mathtt{map}_{\phi_{res}}(q_j, q_{j+1}) = \mathtt{var}$ by the sequence of nodes: (1) $q_j q_{2m-1} q_{2m} q_{j+1}$ with $\mathtt{map}_\phi(q_j, q_{2m-1}) = \mathtt{var}$, $\mathtt{map}_\phi(q_{2m-1}, q_{2m}) = \mathtt{rel}$ and $\mathtt{map}_\phi(q_{2m}, q_{j+1}) = \mathtt{var}$, or (2) $q_j q_{2m} q_{2m-1} q_{j+1}$ with $\mathtt{map}_\phi(q_j, q_{2m}) = \mathtt{var}$, $\mathtt{map}_\phi(q_{2m}, q_{2m-1}) = \mathtt{rel}$ and $\mathtt{map}_\phi(q_{2m-1}, q_{j+1}) = \mathtt{var}$. Thus, we can remove all the newly edges and replace them by some edges in G_ϕ. Hence we can obtain a sequence of nodes $p_0, p_1 \ldots, p_r$ of G_ϕ such that such that 1) for every $i : 0 \leq i \leq r$, $\mathtt{map}(p_i, p_{i+1\%(r+1)})$ is defined, and 2) for each $i : 0 \leq i < r$, $\mathtt{map}(p_i, p_{i+1}) \neq \mathtt{map}(p_{i+1}, p_{i+2\%(r+1)})$.

Now, we have two cases:

- We have $p_i \neq p_j$ for all $i, j \in \{0, \ldots, r\}$ such that $i \neq j$ and hence $p_0, p_1 \ldots, p_r$ is a dependency cycle in G_ϕ (which a contradiction).
- There are $i, j \in \{1, \ldots, r\}$ such that $i < j$ such that $p_i = p_j$ and for every $i', j' \in \{i, \ldots, j-1\}$ such that $i' \neq j'$, we have $p_{j'} \neq p_{i'}$. Now we can show that $p_i p_{i+1} \ldots, p_{j-1}$ is a dependency graph in G_ϕ which also contradict the acyclicity of ϕ. Observe that $j > i + 1$ otherwise it is a contradiction of the linearity of ϕ.

SPLIT'$_{\text{EQ-VAR}}$: We will show the claim for the SPLIT'$_{\text{EQ-VAR}}$.

Let first concentrate on the case SPLIT'$_{\text{EQ-VAR-1}}$ (the other case SPLIT'$_{\text{EQ-VAR-2}}$ can't be applied due to the acyclic formula since the variable u can't appear on both sides).

Let us assume that the equality $e_m : tr_m \neq tr'_m$, with $tr_m = u \cdot tr$ and $tr'_m = tr' \cdot v \cdot tr''$, is transformed into $tr = v_2 \cdot tr''$. Then, the resulting formula ϕ_{res} is of the form $\phi'[u/tr' \cdot v_1][v/v_1 \cdot v_2] \wedge (\bigwedge_{i=1}^{m-1}(tr_i[u/tr' \cdot v_1][v/v_1 \cdot v_2] \approx_i tr'_i[u/tr' \cdot v_1][v/v_1 \cdot v_2])) \wedge (tr = v_2 \cdot tr'')$.

The application of a such rule may be seen as two consecutive steps which all preserve acyclic form:

1. removal of v and its replacement by $v_1 v_2$ in ϕ. We obtain the formula ϕ_1 which is of the form $\phi'[v/v_1 v_2] \wedge (\bigwedge_{i=1}^{m}(e_i[v/v_1 v_2]))$.
2. Then, we apply the SPLIT$_{\text{EQ-VAR-1}}$ to ϕ_1 to obtain ϕ_{res}.

It is clear that ϕ_1 is an acyclic formula and hence ϕ_{res} from the previous item.

10 Proof of Theorem 2

We need only to show that repeatedly applying EQ-VAR, EQ-WORD and EQ to the same equality will terminate. From Theorem 1, we know that after each application of such rules, the acyclicity of the formula is persevered and hence its linearity. This implies that after application of such rules the size of the equality (i.e., the number of variables and constants) will be strictly smaller.