Twinning Automata and Regular Expressions for String Static Analysis

Twinning automata and regular expressions for string static analysis

Luca Negrini[1,2], Vincenzo Arceri[1]✉, Pietro Ferrara[1], and Agostino Cortesi[1]

[1] Ca' Foscari University of Venice, Venice, Italy
{vincenzo.arceri|pietro.ferrara|cortesi}@unive.it
[2] JuliaSoft S.r.l., Verona, Italy
luca.negrini@unive.it

Abstract. In this paper we formalize Tarsis, a new abstract domain based on the abstract interpretation theory that approximates string values through finite state automata. The main novelty of Tarsis is that it works over an alphabet of strings instead of single characters. On the one hand, such an approach requires a more complex and refined definition of the widening operator, and the abstract semantics of string operators. On the other hand, it is in position to obtain strictly more precise results than state-of-the-art approaches. We implemented a prototype of Tarsis, and we applied it to some case studies taken from some of the most popular Java libraries manipulating string values. The experimental results confirm that Tarsis is in position to obtain strictly more precise results than existing analyses.

Keywords: String analysis · Static analysis · Abstract interpretation.

1 Introduction

Strings play a key role in any programming language due to the many and different ways in which they are used, for instance to dynamically access object properties, to hide the program code by using string-to-code statements and reflection, or to manipulate data-interchange formats, such as JSON, just to name a few. Despite the great effort spent in reasoning about strings, static analysis often failed to manage programs that heavily manipulate strings, mainly due to the inaccuracy of the results and the prohibitive amount of resources (time, space) required to retrieve useful information on strings. On the one hand, finite height string abstractions [16] are computable in a reasonable time, but precision is suddenly lost when using advanced string manipulations. On the other hand, more sophisticated abstractions (e.g., the ones reported in [8, 14]) compute precise results but they require a huge, and sometimes unrealistic, computational cost, making such code intractable for these abstractions. A good representation of such abstractions is the finite state automata domain [8]. Over-approximating strings into finite state automata has shown to increase string analysis accuracy in many scenarios, but it does not scale up to real world programs dealing with statically unknown inputs and long text manipulations.

In this paper we introduce Tarsis, a new abstract domain for string values based on finite state automata (FSA). Standard FSA has been shown to provide precise abstractions of string values when all the components of such strings are known, but with high computational cost. Instead of considering standard finite automata built over an alphabet of single characters, Tarsis considers automata that are built over an alphabet of strings. The alphabet comprises a special value to represent statically unknown strings. This avoids the creation of self-loops with any possible character as input, which otherwise would significantly degrade performance. We define the abstract semantics of mainstream string operations, namely substring, length, indexOf, replace, concat and contains, either defined directly on the automaton or on its equivalent regular expression.

Tarsis has been implemented into a prototypical static analyzer supporting a subset of Java. By comparing Tarsis with other cutting-edge domains for string analysis, results show that (i) when applied to simple code that causes a precision loss in simpler domains, Tarsis correctly approximates string values within a comparable execution time, (ii) on code that makes the standard automata domain unusable due to the complexity of the analysis, Tarsis is in position to perform in a limited amount of time, making it a viable domain for complex and real codebases, and (iii) Tarsis is able to precisely abstract complex string operations that have not been addressed by state-of-the-art domains.

The rest of the paper is structured as follows. Sect. 2 introduces a motivating example. Sect. 3 defines the mathematical notation used throughout the paper. Sect. 4 formalizes Tarsis and its abstract semantics. Sect. 5 reports experimental results and comparison with other domains, while Sect. 6 concludes.

1.1 Related work

The problem of statically analyzing strings has been already tackled in different contexts in the literature [14, 8, 29, 13, 25, 2, 16]. The original finite state automata abstract domain was defined in [8] in the context of dynamic languages, providing an automata-based abstract semantics for common ECMAScript string operations. The same abstract domain has been integrated also for defining a sound-by-construction analysis for string-to-code statements [7]. The authors of [4] provided an automata abstraction merged with interval abstractions for analyzing JavaScript arrays and objects. In [13], the authors proposed a static analysis of Java strings based on the abstraction of the control-flow graph as a context-free grammar. Regular strings [12] is an abstraction of the finite state automata domain and approximates strings as a strict subset of regular expressions. Even if it is does not tackle the problem of analyzing strings, in [28] a lattice-based generalization of regular expressions was proposed, showing a regular expressions-based domain parametric from a lattice of reference. An interesting automata-based model is symbolic automata [21], that differs from the standard one having an alphabet of predicates (that can potentially be infinite) instead of single characters. Examples of applications of symbolic automata in the context of static analysis are regex processing, sanitizer analysis [32] and their usage as program model for mixing syntactic and semantic abstractions

```
1  int countMatches(String str, String sub) {
2    int count = 0;
3    int len = sub.length();
4    while (str.contains(sub)) {
5      int idx = str.indexOf(sub);
6      count = count + 1;
7      int start = idx + len;
8      int end = str.length();
9      str = str.substring(start, end);
10   }
11   return count;
12 }
```

Fig. 1: A program that counts the occurrences of a string into another one

over the program [30]. Finally, orthogonally to static analysis of strings by abstract interpretation, a big effort was spent in the context of string constraints verification, focusing on the study of decidable fragments of the string constraint formulas [3] or proposing new efficient decidable procedures or string constraints representations [3, 11, 5] also based on automata, such as [33, 34], or involving type conversion string constraints [1].

2 Motivating example

Consider the code of Fig. 1 that counts the occurrences of string sub into string str. This code is (a simplification of) the *Apache commons-lang* library method StringUtils.countMatches[3], one of the most popular Java libraries providing extra functionalities over the core classes of the Java *lang* package (that contains class String as well). Proving properties about the value of count after the loop is particularly challenging, since it requires to correctly model a set of string operations (namely length, contains, indexOf, and substring) and their interaction. State-of-the-art string analyses fail to precisely model most of such operations, since their abstraction of string values is not rigorous enough to deal with such situations. This loss of precision usually leads to failure in proving string-based properties (also on non-string values) in real-world software, such as the numerical bounds of the value returned by countMatches when applied to a string.

The goal of this paper is to provide an abstract interpretation-based static analysis, in order to deal with complex and nested string manipulations similar to the one reported in Fig. 1. As we will discuss in Sect. 5, Tarsis models (among the others) all string operations used in countMatches, and it is precise enough to infer, given the abstractions of str and sub, the precise range of values that count might have at the end of the method.

3 Preliminaries

Mathematical notation. Given a set S, S^* is the set of all finite sequences of elements of S. If $s = s_0 \dots s_n \in S^*$, s_i is the i-th element of s, $|s| = n + 1$ is its length, and $s[x/y]$ is the sequence obtained replacing all occurrences of x

[3] `https://commons.apache.org/proper/commons-lang/`

in s with y. When s' is a subsequence of s, we write $s' \curvearrowright_{\mathtt{s}} s$. We denote by $s^n, n \geq 0$ the n-times repetition of the string s. Given two sets S and T, $\wp(S)$ is the powerset of S, $S \setminus T$ is the set difference, $S \subset T$ is the strict inclusion relation between S and T, $S \subseteq T$ is the inclusion relation between S and T, and $S \times T$ is the Cartesian product between S and T.

Ordered structures. A set L with a partial ordering relation $\leq \subseteq L \times L$ is a poset, denoted by $\langle L, \leq \rangle$. A poset $\langle L, \leq, \vee, \wedge \rangle$, where $\vee$ and $\wedge$ are respectively the least upper bound (lub) and greatest lower bound (glb) operators of L, is a lattice if $\forall x, y \in L \,.\, x \vee y$ and $x \wedge y$ belong to L. It is also complete if $\forall X \subseteq L$ we have that $\bigvee X, \bigwedge X \in L$. A complete lattice L, with ordering $\leq$, lub $\vee$, glb $\wedge$, top element $\top$, and bottom element $\bot$ is denoted by $\langle L, \leq, \vee, \wedge, \top, \bot \rangle$.

Abstract interpretation. Abstract interpretation [17, 18] is a theoretical framework for sound reasoning about semantic properties of a program, establishing a correspondence between the concrete semantics of a program and an approximation of it, called abstract semantics. Let C and A be complete lattices, a pair of monotone functions $\alpha : C \to A$ and $\gamma : A \to C$ forms a *Galois Connection* (GC) between C and A if $\forall x \in C, \forall y \in A : \alpha(x) \leq_A y \Leftrightarrow x \leq_C \gamma(y)$. We denote a GC as $C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A$. Given $C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A$, a concrete function $f : C \to C$ is, in general, not computable. Hence, a function $f^\sharp : A \to A$ that must *correctly* approximate the function f is needed. If so, we say that the function $f^\sharp$ is *sound*. Given $C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A$ and a concrete function $f : C \to C$, an abstract function $f^\sharp : A \to A$ is sound w.r.t. f if $\forall c \in C. \, \alpha(f(c)) \leq_A f^\sharp(\alpha(c))$. Completeness [24] can be obtained by enforcing the equality of the soundness condition and it is called *backward completeness*. Given $C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A$, a concrete function $f : C \to C$ and an abstract function $f^\sharp : A \to A$, $f^\sharp$ is backward complete w.r.t. f if $\forall c \in C. \, \alpha(f(c)) = f^\sharp(\alpha(c))$.

Finite state automata and regular expression notation. We follow the notation reported in [8] for introducing finite state automata. A finite state automaton (FA) is a tuple $\mathtt{A} = \langle Q, \Sigma, \delta, q_0, F \rangle$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, Σ is a finite alphabet of symbols, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation and $F \subseteq Q$ is the set of final states. If $\delta : Q \times \Sigma \to Q$ is a function then $\mathtt{A}$ is called deterministic finite state automaton. The set of all the FAs is FA. If $\mathscr{L} \subseteq \Sigma^*$ is recognized by a FA, we say that $\mathscr{L}$ is a regular language. Given $\mathtt{A} \in$ FA, $\mathscr{L}(\mathtt{A})$ is the language accepted by $\mathtt{A}$. From the Myhill-Nerode theorem, for each regular language uniquely exists a minimum FA (w.r.t. the number of states) recognizing the language. Given a regular language $\mathscr{L}$, $\mathsf{Min}(\mathtt{A})$ is the minimum FA $\mathtt{A}$ s.t. $\mathscr{L} = \mathscr{L}(\mathtt{A})$. Abusing notation, given a regular language $\mathscr{L}$, $\mathsf{Min}(\mathscr{L})$ is the minimal FA recognizing $\mathscr{L}$. We denote as $\mathsf{paths}(\mathtt{A}) \in \wp(\delta^*)$ the set of sequences of transitions corresponding to all the possible paths from the initial state q_0 to a final state $q_n \in F$. When $\mathtt{A}$ is cycle-free, the set $\mathsf{paths}(\mathtt{A})$ is finite and computable. Given $\pi \in \mathsf{paths}(\mathtt{A})$, $|\pi|$ is its length, meaning the sum of the lengths of the symbols that appear on the transitions composing the path. Furthermore, $|\mathsf{minPath}(\mathtt{A})| \in \mathbb{N}$ denotes the (unique) length of a minimum path. If $\mathtt{A}$ is a cycle-free automaton, $|\mathsf{maxPath}(\mathtt{A})| \in \mathbb{N}$ denotes the (unique) length of a maximum path. Given $\pi = t_0 \dots t_n \in \mathsf{paths}(\mathtt{A})$, σ_{π_i} is the symbol read by the

$$
\begin{array}{l}
\texttt{a} \in \textsc{ae} ::= x \in \textsc{Id} \mid n \in \mathbb{Z} \mid \texttt{a + a} \mid \texttt{a - a} \mid \texttt{a * a} \mid \texttt{a / a} \\
\quad \mid \texttt{length(s)} \mid \texttt{indexOf(s,s)} \\
\texttt{b} \in \textsc{be} ::= x \in \textsc{Id} \mid \texttt{true} \mid \texttt{false} \mid \texttt{b \&\& b} \mid \texttt{b || b} \mid \texttt{! b} \\
\quad \mid \texttt{e} < \texttt{e} \mid \texttt{e == e} \mid \texttt{contains}(\texttt{s}_1\texttt{,s}_2\texttt{)} \\
\texttt{s} \in \textsc{se} ::= x \in \textsc{Id} \mid \text{"}\sigma\text{"} \mid \texttt{substr(s,a,a)} \\
\quad \mid \texttt{concat(s,s)} \mid \texttt{replace(s,s,s)} \quad (\sigma \in \Sigma^*) \\
\texttt{e} \in \textsc{e} ::= \texttt{a} \mid \texttt{b} \mid \texttt{s} \\
\texttt{st} \in \textsc{stmt} ::= \texttt{st ; st} \mid \texttt{skip} \mid x \texttt{ = e} \mid \texttt{if (b) \{st\} else \{st\}} \\
\quad \mid \texttt{while (b) \{st\}} \\
\texttt{P} \in \textsc{Imp} ::= \texttt{st ;}
\end{array}
$$

Fig. 2: Imp syntax

transition t_i, $i \in [0, n]$, and $\sigma_\pi = \sigma_{\pi_0} \dots \sigma_{\pi_n}$ is the string recognized by such path. Predicate $\mathsf{cyclic}(\texttt{A})$ holds if and only if the given automaton contains a cycle. Throughout the paper, it could be more convenient to refer to a finite state automaton by its regular expression (regex for short), being equivalent. Given two regexes $\texttt{r}_1$ and $\texttt{r}_2$, $\texttt{r}_1 \mid\mid \texttt{r}_2$ is the disjunction between $\texttt{r}_1$ and $\texttt{r}_2$, $\texttt{r}_1\texttt{r}_2$ is the concatenation of $\texttt{r}_1$ with $\texttt{r}_2$, $(\texttt{r}_1)^*$ is the Kleene-closure of $\texttt{r}_1$.

The finite state automata abstract domain. Here, we report the necessary notions about the finite state automata abstract domain presented in [8], over-approximating string properties as the minimum deterministic finite state automaton recognizing them. Given an alphabet Σ, the finite state automata domain is defined as $\langle \textsc{Fa}_{/\equiv}, \sqsubseteq_{\textsc{Fa}}, \sqcup_{\textsc{Fa}}, \sqcap_{\textsc{Fa}}, \mathsf{Min}(\varnothing), \mathsf{Min}(\Sigma^*) \rangle$, where $\textsc{Fa}_{/\equiv}$ is the quotient set of Fa w.r.t. the equivalence relation induced by language equality, $\sqsubseteq_{\textsc{Fa}}$ is the partial order induced by language inclusion, $\sqcup_{\textsc{Fa}}$ and $\sqcap_{\textsc{Fa}}$ are the lub and the glb, respectively. The minimum is $\mathsf{Min}(\varnothing)$, that is, the automaton recognizing the empty language, and the maximum is $\mathsf{Min}(\Sigma^*)$, that is, the automaton recognizing any possible string over Σ. We abuse notation by representing equivalence classes in $\textsc{Fa}_{/\equiv}$ by one of its automaton (usually the minimum), i.e., when we write $\texttt{A} \in \textsc{Fa}_{/\equiv}$ we mean $[\texttt{A}]_\equiv$. Since $\textsc{Fa}_{/\equiv}$ does not satisfy the Ascending Chain Condition (ACC), i.e., it contains infinite ascending chains, it is equipped with the parametric widening $\nabla^n_{\textsc{Fa}}$. The latter is defined in terms of a state equivalence relation merging states that recognize the same language, up to a fixed length $n \in \mathbb{N}$, a parameter used for tuning the widening precision [10, 23]. For instance, let us consider the automata $\texttt{A}, \texttt{A}' \in \textsc{Fa}_{/\equiv}$ recognizing the languages $\mathscr{L} = \{\epsilon, a\}$ and $\mathscr{L}' = \{\epsilon, a, aa\}$, respectively. The result of the application of the widening $\nabla^n_{\textsc{Fa}}$, with $n = 1$, is $\texttt{A} \nabla^n_{\textsc{Fa}} \texttt{A}' = \texttt{A}''$ s.t. $\mathscr{L}(\texttt{A}'') = \{\ a^n \mid n \in \mathbb{N}\ \}$.

Core language and semantics. We introduce a minimal core language Imp, whose syntax is reported in Fig. 2. Such language supports the main operators over strings. In particular, Imp supports arithmetic expressions (ae), Boolean expressions (be) and string expressions (se). Primitives values are $\textsc{Val} = \mathbb{Z} \cup \Sigma^* \cup \{\texttt{true}, \texttt{false}\}$, namely integers, strings and booleans. Programs states $\mathbb{M} : \textsc{Id} \to \textsc{Val}$ map identifiers to primitives values, ranged over the meta-variable $\mathtt{m}$. The concrete semantics of Imp statements is captured by the function $[\![\ \texttt{st}\ ]\!]$:

$$
\begin{aligned}
[\![\ \texttt{substr}(\mathsf{s},\mathsf{a},\mathsf{a}')\ ]\!]\mathtt{m} &= \sigma_i \dots \sigma_j \qquad \text{if } i \leq j < |\sigma|, i = [\![\ \mathsf{s}\ ]\!]\mathtt{m}, j = [\![\ \mathsf{s}'\ ]\!]\mathtt{m} \\
[\![\ \texttt{length}(\mathsf{s})\ ]\!]\mathtt{m} &= |\sigma| \\
[\![\ \texttt{indexOf}(\mathsf{s},\mathsf{s}')\ ]\!]\mathtt{m} &= \begin{cases} \min\{\ i \mid \sigma_i \dots \sigma_j = \sigma'\ \} & \text{if } \exists i,j \in \mathbb{N}\,.\,\sigma_i \dots \sigma_j = \sigma' \\ -1 & \text{otherwise} \end{cases} \\
[\![\ \texttt{replace}(\mathsf{s},\mathsf{s}',\mathsf{s}'')\ ]\!]\mathtt{m} &= \begin{cases} \sigma[\sigma'/\sigma''] & \text{if } \sigma' \curvearrowright_{\mathsf{s}} \sigma \\ \sigma & \text{otherwise} \end{cases} \\
[\![\ \texttt{concat}(\mathsf{s},\mathsf{s}')\ ]\!]\mathtt{m} &= \sigma \cdot \sigma' \\
[\![\ \texttt{contains}(\mathsf{s},\mathsf{s}')\ ]\!]\mathtt{m} &= \begin{cases} \texttt{true} & \text{if } \exists i,j \in \mathbb{N}\,.\,\sigma_i \dots \sigma_j = \sigma' \\ \texttt{false} & \text{otherwise} \end{cases} \\
\text{where } \sigma = [\![\ \mathsf{s}\ ]\!]\mathtt{m}, \sigma' &= [\![\ \mathsf{s}'\ ]\!]\mathtt{m}, \sigma'' = [\![\ \mathsf{s}''\ ]\!]\mathtt{m}
\end{aligned}
$$

Fig. 3: Concrete semantics of Imp string expressions

$\mathbb{M} \to \mathbb{M}$. The semantics is defined in a standard way and for this reason has been omitted. Such semantics relies on the one of expressions, that we capture, abusing notation, as $[\![\ \mathsf{e}\ ]\!] : \mathbb{M} \to \textsc{Val}$. While the semantics concerning arithmetic and Boolean expressions is straightforward (and not of interest of this paper), we define the part concerning strings in Fig. 3.

4 The Tarsis abstract domain

In this section, we recast the original finite state abstract domain working over an alphabet of characters Σ, reported in Sect. 3, to an augmented abstract domain based on finite state automata over an alphabet of strings.

4.1 Abstract domain and widening

The key idea of Tarsis is to adopt the same abstract domain, changing the alphabet on which finite state automata are defined to a set of strings, namely Σ^*. Clearly, the main concern here is that Σ^* is infinite and this would not permit us to adopt the finite state automata model, that requires the alphabet to be finite. Thus, in order to solve this problem, we make this abstract domain *parametric* to the program we aim to analyze and in particular to its strings. Given an Imp program P, we denote by Σ^*_{P} any substring of strings appearing in P, [4] *delimiting* the space of string properties we aim to check only on P.

At this point, we can instantiate the automata-based framework proposed in [8] with the new alphabet as

$$\langle \mathcal{T}\mathrm{FA}_{/\equiv}, \sqsubseteq_{\mathcal{T}}, \sqcup_{\mathcal{T}}, \sqcap_{\mathcal{T}}, \mathsf{Min}(\varnothing), \mathsf{Min}(\mathbb{A}^*_{\mathsf{P}}) \rangle$$

[4] The set Σ^*_{P} can be easily computed collecting the constant strings in P by visiting its abstract syntax tree and then computing their substrings.

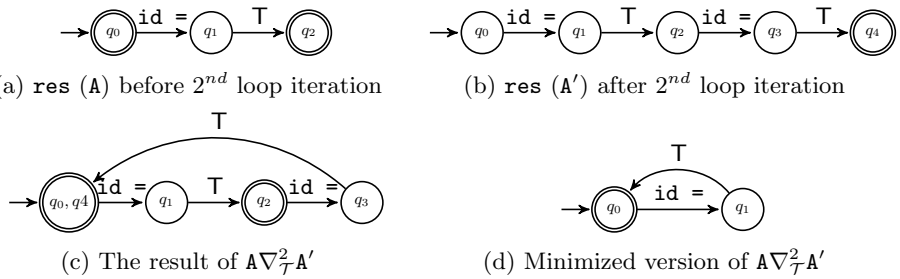
The alphabet on which finite state automata are defined is $\mathbb{A}_{\mathsf{P}} \triangleq \Sigma^*_{\mathsf{P}} \cup \{\mathsf{T}\}$, where T is a special symbol that we intend as *"any possible string"*. Let $\mathcal{T}\text{FA}$ be the set of any deterministic finite state automaton over the alphabet $\mathbb{A}_{\mathsf{P}}$. Since we can have more automata recognizing a language, $\mathcal{T}\text{FA}_{/\equiv}$ is the quotient set of $\mathcal{T}\text{FA}$ w.r.t. the equivalence relation induced by language equality, that is, the elements of domain are equivalence classes. For simplicity, when we write $\mathtt{A} \in \mathcal{T}\text{FA}_{/\equiv}$, we intend the equivalence class of $\mathtt{A}$. $\sqsubseteq_{\mathcal{T}}$ is the partial order induced by language inclusion, $\sqcup_{\mathcal{T}}$ and $\sqcap_{\mathcal{T}}$ are the lub and the glb over elements of $\mathcal{T}\text{FA}_{/\equiv}$, computing the equivalence class of the union and the intersection of the two automata representing the corresponding classes, respectively. The bottom element is $\mathsf{Min}(\varnothing)$, corresponding to the automaton recognizing the empty language, and the maximum is $\mathsf{Min}(\mathbb{A}^*_{\mathsf{P}})$, namely the automaton recognizing any string over $\mathbb{A}_{\mathsf{P}}$.

Like in the standard finite state automata domain $\text{FA}_{/\equiv}$, also $\mathcal{T}\text{FA}_{/\equiv}$ is not a complete lattice and, consequently, it does not form a Galois Connection with the string concrete domain $\wp(\Sigma^*)$. This comes from the non-existence, in general, of the best abstraction of a string set in $\mathcal{T}\text{FA}_{/\equiv}$ (e.g., a context-free language has no best abstract element in $\mathcal{T}\text{FA}_{/\equiv}$ approximating it). Nevertheless, this is not a concern since weaker forms of abstract interpretation are still possible [19] still guaranteeing soundness relations between concrete and abstract elements (e.g., polyhedra [20]). In particular, we can still ensure soundness comparing the concretizations of our abstract elements (cf. Sect. 8 of [19]). Hence, we define the concretization function $\gamma_{\mathcal{T}} : \mathcal{T}\text{FA}_{/\equiv} \to \wp(\Sigma^*)$ as $\gamma_{\mathcal{T}}(\mathtt{A}) \triangleq \bigcup_{\sigma \in \mathscr{L}(\mathtt{A})} \mathsf{Flat}(\sigma)$, where Flat converts a string over $\mathbb{A}_{\mathsf{P}}$ into a set of strings over Σ^*. For instance, $\mathsf{Flat}(a\ \mathsf{T}\mathsf{T}\ bb\ c) = \{\ a\sigma bbc \mid \sigma \in \Sigma^*\ \}$. Note that, the language of strings (over the alphabet Σ) recognized by $\mathtt{A}$ corresponds to the concretization function reported above, namely $\mathscr{L}(\mathtt{A}) = \gamma_{\mathcal{T}}(\mathtt{A})$.

Widening. Similarly to the standard automata domain $\text{FA}_{/\equiv}$, also $\mathcal{T}\text{FA}_{/\equiv}$ does not satisfy ACC, meaning that fix-point computations over $\mathcal{T}\text{FA}_{/\equiv}$ may not converge in a finite time. Hence, we need to equip $\mathcal{T}\text{FA}_{/\equiv}$ with a widening operator to ensure the convergence of the analysis. We define the widening operator $\nabla^n_{\mathcal{T}} : \mathcal{T}\text{FA}_{/\equiv} \times \mathcal{T}\text{FA}_{/\equiv} \to \mathcal{T}\text{FA}_{/\equiv}$, parametric in $n \in \mathbb{N}$, taking two automata as input and returning an over-approximation of the least upper bounds between them, as required by widening definition. We rely on the standard automata widening reported in Sect. 3, that, informally speaking, can be seen as a *subset construction* algorithm [22] up to languages of strings of length n. In order to explain the widening $\nabla^n_{\mathcal{T}}$, consider the following function manipulating strings.[5]

```
1  function f(v) {
2      res = "";
3      while (?)
4          res = res + "id = " +  v;
5      return res;
6  }
```

[5] For the sake of readability, in the program examples presented in this paper the plus operation between strings corresponds to the string concatenation.



(a) `res` (`A`) before 2^{nd} loop iteration

(b) `res` (`A'`) after 2^{nd} loop iteration

(c) The result of $\mathtt{A}\nabla^2_{\mathcal{T}}\mathtt{A}'$

(d) Minimized version of $\mathtt{A}\nabla^2_{\mathcal{T}}\mathtt{A}'$

Fig. 4: Example of widening application

The function `f` takes as input parameter `v` and returns variable `res`. Let us suppose that `v` is a statically unknown string, corresponding to the automaton recognizing T (i.e., $\mathsf{Min}(\{\mathsf{T}\})$). The result of the function `f` is a string of the form `id` $=\mathsf{T}$, repeated zero or more times. Since the `while` guard is unknown, the number of iterations is statically unknown, and in turn, also the number of performed concatenations inside the loop body. The goal here is to over-approximate the value returned by the function `f`, i.e., the value of `res` at the end of the function.

Let `A`, reported in Fig. 4a, be the automaton abstracting the value of `res` before starting the second iteration of the loop, and let `A'`, reported in Fig. 4b be the automaton abstracting the value of `res` at the end of the second iteration. At this point, we want to apply the widening operator $\nabla^n_{\mathcal{T}}$, between `A` and `A'`, working as follows. We first compute $\mathtt{A} \sqcup_{\mathcal{T}} \mathtt{A}'$ (corresponding to the automaton reported in Fig. 4b except that also q_0 and q_2 are final states). On this automaton, we merge any state that recognizes the same $\mathbb{A}_{\mathsf{P}}$-strings of length n, with $n \in \mathbb{N}$. In our example, let n be 2. The resulting automaton is reported in Fig. 4c, where q_0 and q_4 are put together, the other states are left as singletons since they cannot be merged with no other state. Fig. 4d depicts the minimized version of Fig. 4c.

The widening $\nabla^n_{\mathcal{T}}$ has been proved to meet the widening requirements (i.e., over-approximation of the least upper bounds and convergence on infinite ascending chains) in [23]. The parameter n, tuning the widening precision, is arbitrary and can be chosen by the user. As highlighted in [8], the higher n is, the more the corresponding widening operator is precise in over-approximating lubs of infinite ascending chains (i.e., in fix-point computations).

A classical improvement on widening-based fix-point computations is to integrate a threshold [15], namely widening is applied to over-approximate lubs when a certain threshold (usually over some property of abstract values) is overcome. In fix-point computations, we decide to apply the previously defined widening $\nabla^n_{\mathcal{T}}$ only when the number of the states of the lubbed automata overcomes the threshold $\tau \in \mathbb{N}$. This permits us to postpone the widening application, getting more precise abstractions when the automata sizes do not overcome the threshold. At the moment, the threshold τ is not automatically inferred, since it surely requires further investigations.

4.2 String abstract semantics of IMP

In this section, we define the abstract semantics of the string operators defined in Sect. 3 over the new string domain $\mathcal{T}\text{FA}_{/\equiv}$. Since IMP supports strings, integers and Booleans values, we need a way to merge the corresponding abstract domains. In particular, we abstract integers with the well-known interval abstract domain [17] defined as $\mathsf{Intv} \triangleq \{\ [a,b] \mid a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{+\infty\}, a \leq b\ \} \cup \{\bot_{\mathsf{Intv}}\}$ and Booleans with $\mathsf{Bool} \triangleq \wp(\{\mathtt{true}, \mathtt{false}\})$. As usual, we denote by $\sqcup_{\mathsf{Intv}}$ and $\sqcup_{\mathsf{Bool}}$ the lubs between intervals and Booleans, respectively. In particular, we merge such abstract domains in $\text{VAL}^\sharp$ by the coalesced sum abstract domain [6] as

$$\text{VAL}^\sharp \triangleq \mathcal{T}\text{FA}_{/\equiv} \oplus \mathsf{Intv} \oplus \mathsf{Bool}$$

Informally, the coalesced sum abstract domain introduces a new bottom and top element, and it *coalesces* the bottom elements of the involved domains.

The program state is represented through abstract program memories $\mathbb{M}^\sharp : \text{ID} \to \text{VAL}^\sharp$ from identifiers to abstract values. The abstract semantics is captured by the function $⟅\mathtt{st}⟆ : \mathbb{M}^\sharp \to \mathbb{M}^\sharp$, relying on the abstract semantics of expressions defined by, abusing notation, $⟅\mathtt{e}⟆ : \mathbb{M}^\sharp \to \text{VAL}^\sharp$. We focus on the abstract semantics of string operations[6], while the semantics of the other expressions is standard and does not involve strings.

In order to define the abstract semantics of IMP over TARSIS, it is worth to highlight that one can think to reuse the one adopted in the standard finite state automata abstract domain [8]: unfortunately, this is not possible since the one reported in [8] only deals with automata over alphabet of single characters (not strings), and does not handle the character T used in TARSIS alphabet, that must be treated, as we will see soon, as a special symbol.

Length. Given $\mathtt{A} \in \mathcal{T}\text{FA}_{/\equiv}$, the abstract semantics of `length` returns an interval $[c_1, c_2]$ such that $\forall \sigma \in \mathscr{L}(\mathtt{A})\,.\, c_1 \leq |\sigma| \leq c_2$. We recast the original idea of the abstract semantics of `length` over standard finite state automata. Let $\mathtt{s} \in \text{SE}$, supposing that $⟅\mathtt{s}⟆\mathfrak{m}^\sharp = \mathtt{A} \in \mathcal{T}\text{FA}_{/\equiv}$. The `length` abstract semantics is:

$$⟅\mathtt{length(s)}⟆\mathfrak{m}^\sharp \triangleq \begin{cases} [|\mathsf{minPath}(\mathtt{A})|, +\infty] & \text{if } \mathsf{cyclic}(\mathtt{A}) \vee \mathsf{readsTop}(\mathtt{A}) \\ [|\mathsf{minPath}(\mathtt{A})|, |\mathsf{maxPath}(\mathtt{A})|] & \text{otherwise} \end{cases}$$

where $\mathsf{readsTop}(\mathtt{A}) \Leftrightarrow \exists q, q' \in Q\,.\,(q, \mathsf{T}, q') \in \delta$. Note that, when evaluating the length of the minimum path, T is considered to have a length of 0. For instance, consider the automaton `A` reported in Fig. 5a. The minimum path of `A` is $(q_0, aa, q_1), (q_1, \mathsf{T}, q_2), (q_2, bb, q_4)$ and its length is 4. Since a transition labeled with T is in `A` (and its length cannot be statically determined), the abstract `length` of `A` is $[4, +\infty]$. Consider the automaton $\mathtt{A}'$ reported in Fig. 5b. In this case, $\mathtt{A}'$ has no cycles and has no transitions labeled with T and the length of any string recognized by $\mathtt{A}'$ can be determined. The length of the minimum path of $\mathtt{A}'$ is 3 (below path of $\mathtt{A}'$), the length of the maximum path of $\mathtt{A}'$ is 7 (above path of $\mathtt{A}'$) and consequently the abstract `length` of $\mathtt{A}'$ is $[3, 7]$.

[6] The abstract semantics of `concat` does not add any further important technical detail to the paper hence it is not reported.

(a) (b)

Fig. 5: (a) $\mathtt{A}$ s.t. $\mathscr{L}(\mathtt{A}) = \{bbb\ bbb, aa\ \top\ bb\}$, (b) $\mathtt{A}'$ s.t. $\mathscr{L}(\mathtt{A}') = \{a\ b\ c, aa\ bbb\ cc\}$

Contains. Given $\mathtt{A}, \mathtt{A}' \in \mathcal{T}\text{FA}_{/\equiv}$, the abstract semantics of `contains` should return `true` if any string of $\mathtt{A}'$ is surely contained into some string of $\mathtt{A}$, `false` if no string of $\mathtt{A}'$ is contained in some string of $\mathtt{A}$ and $\{\mathtt{true}, \mathtt{false}\}$ in the other cases. For instance, consider the automaton $\mathtt{A}$ depicted in Fig. 6a and suppose we check if it contains the automaton $\mathtt{A}'$ recognizing the language $\{aa, a\}$. The automaton $\mathtt{A}'$ is a *single-path automaton* [9], meaning that any string of $\mathtt{A}'$ is a prefix of its longest string. In this case, the containment of the longest string (on each automaton path) implies the containment of the others, such as in our example, namely it is enough to check that the longest string of $\mathtt{A}'$ is contained into $\mathtt{A}$. Note that, a single-path automaton cannot read the symbol $\top$. We rely on the predicate $\mathsf{singlePath}(\mathtt{A})$ when $\mathtt{A}$ is a non-cyclic single-path automaton and we denote by σ_{sp} its longest string. Let $\mathtt{s}, \mathtt{s}' \in \text{SE}$, supposing that $\wr \mathtt{s} \int \mathfrak{m}^\sharp = \mathtt{A} \in \mathcal{T}\text{FA}_{/\equiv}$, $\wr \mathtt{s}' \int \mathfrak{m}^\sharp = \mathtt{A}' \in \mathcal{T}\text{FA}_{/\equiv}$. The `contains` abstract semantics is:

$$\wr \mathtt{contains}(\mathtt{s}, \mathtt{s}') \int \mathfrak{m}^\sharp \triangleq \begin{cases} \mathtt{false} & \text{if } \mathtt{A}' \sqcap_{\mathcal{T}} \mathsf{FA}(\mathtt{A}) = \mathsf{Min}(\varnothing) \\ \mathtt{true} & \text{if } \mathsf{singlePath}(\mathtt{A}') \\ & \wedge \forall \pi \in \mathsf{paths}(\mathtt{A}^{ac}) \,.\, \sigma_{\mathsf{sp}} \curvearrowright_{\mathtt{s}} \sigma_\pi \\ \{\mathtt{true}, \mathtt{false}\} & \text{otherwise} \end{cases}$$

In the first case, we denote by $\mathsf{FA}(\mathtt{A})$ the factor automaton of $\mathtt{A}$, i.e., the automaton recognizing any substring of $\mathtt{A}$. In particular, if $\mathtt{A}$ does not share any substring of $\mathtt{A}'$, the abstract semantics safely returns `false` (checking the emptiness of the greatest lower bound between $\mathsf{FA}(\mathtt{A})$ and $\mathtt{A}'$). Then, if $\mathtt{A}'$ is a single path automaton, the abstract semantics returns `true` if any path of $\mathtt{A}^{ac}$ reads the longest string of $\mathtt{A}'$, with $\mathtt{A}^{ac}$ being a copy of $\mathtt{A}$ where all the cycles have been removed. Here, we abuse notation denoting with $\sigma_{\mathsf{sp}} \curvearrowright_{\mathtt{s}} \sigma_\pi$ the fact that σ_{sp} is a substring of each string in $\mathsf{Flat}(\sigma_\pi)$. Otherwise, $\{\mathtt{true}, \mathtt{false}\}$ is returned.

IndexOf. Given $\mathtt{A}, \mathtt{A}' \in \mathcal{T}\text{FA}_{/\equiv}$, the `indexOf` abstract semantics returns an interval of the first indexes of the strings of $\mathscr{L}(\mathtt{A}')$ inside strings of $\mathscr{L}(\mathtt{A})$, recalling that when there exists a string of $\mathscr{L}(\mathtt{A}')$ that is not a substring of at least one string of $\mathscr{L}(\mathtt{A}')$, the resulting interval must take into account -1 as well. Let $\mathtt{s}, \mathtt{s}' \in \text{SE}$ and suppose $\wr \mathtt{s} \int \mathfrak{m}^\sharp = \mathtt{A}$ and $\wr \mathtt{s}' \int \mathfrak{m}^\sharp = \mathtt{A}'$. The abstract semantics of

`indexOf` is defined as:

$$\wr\texttt{indexOf}(\mathsf{s},\mathsf{s}')\wr^{\sharp} \triangleq \begin{cases} [-1,+\infty] & \text{if } \mathsf{cyclic}(\mathtt{A}) \vee \mathsf{cyclic}(\mathtt{A}') \vee \mathsf{readsTop}(\mathtt{A}') \\ [-1,-1] & \text{if } \forall \sigma' \in \mathscr{L}(\mathtt{A}')\ \nexists \sigma \in \mathscr{L}(\mathtt{A})\,.\,\sigma' \curvearrowright_{\mathsf{s}} \sigma \\ \bigsqcup\limits^{\mathsf{Intv}}_{\sigma \in \mathscr{L}(\mathtt{A}')} \mathsf{IO}(\mathtt{A},\sigma) & \text{otherwise} \end{cases}$$

If one of the automata has cycles or the automaton abstracting strings we aim to search for ($\mathtt{A}'$) has a T-transition, we return $[-1,+\infty]$. Moreover, if none of the strings recognized by $\mathtt{A}'$ is contained in a string recognized by $\mathtt{A}$, we can safely return the precise interval $[-1,-1]$ since any string recognized by $\mathtt{A}'$ is never a substring of a string recognized by $\mathtt{A}$.[7] If none of the aforementioned conditions is met, we rely on the auxiliary function $\mathsf{IO} : \mathcal{T}\mathrm{FA}_{/\equiv} \times \Sigma^* \to \mathsf{Intv}$, that, given an automaton $\mathtt{A}$ and a string $\sigma \in \Sigma^*$, returns an interval corresponding to the possible first positions of σ in strings recognized by $\mathtt{A}$. Since $\mathtt{A}'$ surely recognizes a finite language (i.e., has no cycles), the idea is to apply $\mathsf{IO}(\mathtt{A},\sigma)$ to each $\sigma \in \mathscr{L}(\mathtt{A}')$ and to return the upper bound of the resulting intervals. In particular, the function $\mathsf{IO}(\mathtt{A},\sigma)$ returns an interval $[i,j] \in \mathsf{Intv}$ where, i and j are computed as follows.

$$i = \begin{cases} -1 & \text{if } \exists \pi \in \mathsf{paths}(\mathtt{A})\,.\,\sigma \not\curvearrowright_{\mathsf{s}} \sigma_\pi \\ \min\limits_{\pi \in \mathsf{paths}(\mathtt{A})} \left\{ i \;\middle|\; \begin{array}{l} \sigma_f \in \mathsf{Flat}(\sigma_\pi) \\ \wedge \sigma_{f_i} \ldots \sigma_{f_{i+n}} = \sigma \end{array} \right\} & \text{otherwise} \end{cases}$$

$$j = \begin{cases} -1 & \text{if } \forall \pi \in \mathsf{paths}(\mathtt{A})\,.\,\sigma \not\curvearrowright_{\mathsf{s}} \sigma_\pi \\ +\infty & \text{if } \exists \pi \in \mathsf{paths}(\mathtt{A})\,.\,\sigma \curvearrowright_{\mathsf{s}} \sigma_\pi \\ & \wedge \pi \text{ reads } \mathsf{T} \text{ before } \sigma \\ \max\limits_{\pi \in \mathsf{paths}(\mathtt{A})} \left\{ i \;\middle|\; \begin{array}{l} \sigma_f \in \mathsf{Flat}(\sigma_\pi) \\ \wedge \sigma_{f_i} \ldots \sigma_{f_{i+n}} = \sigma \\ \wedge \sigma \not\curvearrowright_{\mathsf{s}} \sigma_{f_0} \ldots \sigma_{f_{i+n-1}} \end{array} \right\} & \text{otherwise} \end{cases}$$

As for the abstract semantics of `contains`, we abuse notation denoting with $\sigma \curvearrowright_{\mathsf{s}} \sigma_\pi$ the fact that σ is a substring of each string in $\mathsf{Flat}(\sigma_\pi)$. Given $\mathsf{IO}(\mathtt{A},\sigma) = [i,j] \in \mathsf{Intv}$, i corresponds to the minimal position where the first occurrence of σ can be found in $\mathtt{A}$, while j to the maximal one. Let us first focus on the computation of the minimal position. If there exists a path π of $\mathtt{A}$ s.t. σ is not recognized by σ_π, then the minimal position where σ can be found in $\mathtt{A}$ does not exist and -1 is returned. Otherwise, the minimal position where σ begins across π is returned. Let us consider now the computation of the maximal position. If all paths of the automaton do not recognize σ, then -1 is returned. If there exists a path where σ is recognized but the character T appears earlier in the path,

[7] Note that this is a decidable check since $\mathtt{A}$ and $\mathtt{A}'$ are cycle-free, otherwise the interval $[-1,+\infty]$ would be returned in the first case.

Fig. 6: Example of may-replacement

then $+\infty$ is returned. Otherwise, the maximal index of the first occurrences of σ across the paths of $\mathtt{A}$ is returned.

Replace. In order to give the intuition about how the abstract semantics of `replace` will work, consider the three automata $\mathtt{A}, \mathtt{A}_s, \mathtt{A}_r \in \mathcal{T}\text{FA}_{/\equiv}$. Roughly speaking, the abstract semantics of `replace` substitutes strings of $\mathtt{A}_s$ with strings of $\mathtt{A}_r$ inside strings of $\mathtt{A}$. Let us refer to $\mathtt{A}_s$ as the *search automaton* and to $\mathtt{A}_r$ as the *replace automaton*. We need to specify two types of possible replacements, by means of the following example. Consider $\mathtt{A} \in \mathcal{T}\text{FA}_{/\equiv}$ that is depicted in Fig. 6a and suppose that the search automaton $\mathtt{A}_s$ is the one recognizing the string *bbb* and the replace automaton $\mathtt{A}_r$ is a random automaton. In this case, the `replace` abstract semantics performs a *must-replace* over $\mathtt{A}$, namely substituting the sub-automaton composed by q_1 and q_2 with the replace automaton $\mathtt{A}_r$. Instead, let us suppose that the search automaton $\mathtt{A}_s$ is the one recognizing *bbb* or *cc*. Since it is unknown which string *must* be replaced (between *bbb* and *cc*), the `replace` abstract semantics needs to perform a *may-replace*: when a string recognized by the search automaton is met inside a path of $\mathtt{A}$ it is left unaltered in the automaton and, in the same position where the string is met, the abstract `replace` only extends $\mathtt{A}$ with the replace automaton. An example of may replacement is reported in Fig. 6, where $\mathtt{A}$ is the one reported in Fig. 6a, the search automaton $\mathtt{A}_s$ is the one recognizing the language $\{bbb, cc\}$ and the replace automaton $\mathtt{A}_r$ is the one recognizing the string rr.

Before introducing the abstract semantics of `replace`, we define how to replace a string into an automaton. In particular, we define algorithm RP in Alg. 1, that given $\mathtt{A} \in \mathcal{T}\text{FA}_{/\equiv}$, a replace automaton $\mathtt{A}^r$ and $\sigma \in \Sigma^* \cup \{\mathsf{T}\}$, it returns a new automaton that is identical to $\mathtt{A}$ except that σ is replaced with $\mathtt{A}^r$.

Alg. 1 searches the given string σ across all paths of $\mathtt{A}$, collecting the sequences of transitions that recognize the search string σ and extracting them from the paths of $\mathtt{A}$ (lines 2-3): an ϵ-transition is introduced going from the first state of the sequence to the initial state of $\mathtt{A}'$, and one such transition is also introduced for each final state of $\mathtt{A}'$, connecting that state with the ending state of the sequence (lines 4-5). Then, the list of states composing the sequence of transitions is iterated backwards (lines 6-7), stopping at the first state that has a transition going outside of such list. All the states traversed in this way (excluding the one where the iteration stopped) are removed from the resulting automaton, with the transitions connecting them (lines 8-9), since they were needed only to recognize the string that has been replaced. Note that RP corresponds to a must-replace. At

Algorithm 1: RP algorithm

Data: $\mathtt{A}^o = \langle Q^o, \mathbb{A}, \delta^o, q_0^o, F^o \rangle, \mathtt{A}^r = \langle Q^r, \mathbb{A}, \delta^r, q_0^r, F^r \rangle \in \mathcal{T}\mathrm{FA}_{/\equiv}, \sigma \in \Sigma^* \cup \{\mathsf{T}\}$
Result: $\mathtt{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$

$Q^{result} \leftarrow Q^o \cup Q^r$; $\delta^{result} \leftarrow \delta^o \cup \delta^r$;
foreach $\pi \in \mathsf{paths}(\mathtt{A}^o)$ **do**
  foreach $(q_i, \sigma_0, q_{i+1}), \ldots, (q_{i+n-1}, \sigma_n, q_{i+n}) \in \pi$ **do**
    $\delta^{result} \leftarrow \delta^{result} \cup (q_i, \epsilon, q_0^r)$;
    $\delta^{result} \leftarrow \delta^{result} \cup \{\ (q_f, \epsilon, q_{i+n}) \mid q_f \in F^r\ \}$;
    foreach $k \in [i+n-1, i+1]$ **do**
      if $\nexists (q_k, \sigma', q) \in \delta^o : q \neq q_{k+1}$ **then**
        $Q^{result} \leftarrow Q^{result} \setminus \{q_k\}$;
        $\delta^{result} \leftarrow \delta^{result} \setminus \{(q_k, \sigma', q_{k+1})\}$;
      else break;
return $\langle Q^{result}, \mathbb{A}, \delta^{result}, q_0^o, F^o \rangle$;

this point, we are ready to define the `replace` abstract semantics. In particular, if either $\mathtt{A}$ or $\mathtt{A}_s$ have cycles or $\mathtt{A}_s$ has a T-transition, we return $\mathsf{Min}(\{\mathsf{T}\})$, namely the automaton recognizing T. Otherwise, the `replace` abstract semantics is:

$$\wr\mathtt{replace}(\mathsf{s}, \mathsf{s}_s, \mathsf{s}_r)\int\mathfrak{m}^\sharp \triangleq \begin{cases} \mathtt{A} & \text{if } \forall \sigma_s \in \mathscr{L}(\mathtt{A}_s) \\ & \nexists \sigma \in \mathscr{L}(\mathtt{A})\,. \\ & \sigma_s \curvearrowright_{\mathsf{s}} \sigma \\ \mathsf{RP}(\mathtt{A}, \sigma_s, \mathtt{A}_r) & \text{if } \mathscr{L}(\mathtt{A}_s) = \{\sigma_s\} \\ \bigsqcup\limits_{\sigma \in \mathscr{L}(\mathtt{A}_s)} \mathsf{RP}(\mathtt{A}, \sigma, \mathtt{A}_r \sqcup_{\mathcal{T}} \mathsf{Min}(\{\sigma\})) & \text{otherwise} \end{cases}$$

In the first case, if none of the strings recognized by the search automaton $\mathtt{A}_s$ is contained into strings recognized by $\mathtt{A}$, we can safely return the original automaton $\mathtt{A}$ without any replacement. In the special case where $\mathscr{L}(\mathtt{A}_s) = \{\sigma_s\}$, we return the automaton obtained by performing a replacement calling the function $\mathsf{RP}(\mathtt{A}, \sigma_s, \mathtt{A}_r)$. In the last case, for each each string $\sigma \in \mathscr{L}(\mathtt{A}_s)$, we perform a may replace of σ with $\mathtt{A}_r$: note that, this exactly corresponds to a call RP where the replace automaton is $\mathtt{A}_r \sqcup_{\mathcal{T}} \mathsf{Min}(\{\sigma\})$, namely σ is not removed. The so far obtained automata are finally lubbed together.

Substring. Given $\mathtt{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$ and two intervals $\mathsf{i}, \mathsf{j} \in \mathsf{Intv}$, the abstract semantics of `substring` returns a new automaton $\mathtt{A}'$ soundly approximating any substring from i to j of strings recognized by $\mathtt{A}$, for any $i \in \mathsf{i}, j \in \mathsf{j}$ s.t. $i \leq j$.

Given $\mathtt{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$, in the definition of the `substring` semantics, we rely on the corresponding regex $\mathtt{r}$ since the two representations are equivalent and regexes allow us to define a more intuitive formalization of the semantics of `substring`. Let us suppose that $\wr\mathsf{s}\int\mathfrak{m}^\sharp = \mathtt{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$ and let us denote by $\mathtt{r}$ the regex corresponding to the language recognized by $\mathtt{A}$. At the moment, let us consider exact intervals representing one integer value, namely $\wr\mathsf{a}_1\int\mathfrak{m}^\sharp = [i, i]$ and $\wr\mathsf{a}_2\int\mathfrak{m}^\sharp = [j, j]$, with $i, j \in \mathbb{Z}$. In this case, the abstract semantics is defined as:

$$\wr\mathtt{substr}(\mathsf{s}, \mathsf{a}_1, \mathsf{a}_2)\int\mathfrak{m}^\sharp \triangleq \bigsqcup \mathsf{Min}(\{\ \sigma \mid (\sigma, 0, 0) \in \mathsf{Sb}(\mathtt{r}, i, j-i)\ \})$$

where Sb takes as input a regex $\mathtt{r}$, two indexes $i, j \in \mathbb{N}$, and computes the set of substrings from i to j of all the strings recognized by $\mathtt{r}$. In particular, Sb is

Algorithm 2: Sb algorithm

```
   Data: r regex over 𝔸, i, j ∈ ℕ
   Result: { (σ, n1, n2) | σ ∈ Σ*, n1, n2 ∈ ℕ }
 1 if j = 0 ∨ r = ∅ then
 2 |   return ∅;
 3 else if r = σ ∈ Σ* then
 4 |   if i > |σ| then return {(ϵ, i − |σ|, j)} ;
 5 |   else if i + j > |σ| then return {(σ_i ... σ_{|σ|−1}, 0, j − |σ| + i)} ;
 6 |   else return {(σ_i ... σ_{i+j}, 0, 0)} ;
 7 else if r = ⊤ then
 8 |   result ← {(ϵ, i − k, j) : 0 ≤ k ≤ i, k ∈ ℕ};
 9 |   result ← result ∪ { (•^k, 0, j − k) | 0 ≤ k ≤ j, k ∈ ℕ };
10 |   return result;
11 else if r = r1 r2 then
12 |   result ← ∅;
13 |   subs1 ← Sb(r1, i, j);
14 |   foreach (σ1, i1, j1) ∈ subs1 do
15 |   |   if j1 = 0 then
16 |   |   |   result ← result ∪ {(σ1, i1, j1)};
17 |   |   else
18 |   |   |   result ← result ∪ { (σ1 · σ2, i2, j2) | (σ2, i2, j2) ∈ Sb(r2, i1, j1) };
19 |   return result;
20 else if r = r1||r2 then
21 |   return Sb(r1, i, j) ∪ Sb(r2, i, j);
22 else if r = (r1)* then
23 |   result ← {(ϵ, i, j)}; partial ← ∅;
24 |   repeat
25 |   |   result ← result ∪ partial; partial ← ∅;
26 |   |   foreach (σn, in, jn) ∈ result do
27 |   |   |   foreach (suff, is, js) ∈ Sb(r1, in, in + jn) do
28 |   |   |   |   if ∄(σ′, k, w) ∈ result . σ′ = σn · suff ∧ k = is ∧ w = js then
29 |   |   |   |   |   partial ← partial ∪ {(σn · suff, is, js)};
30 |   until partial ≠ ∅;
31 |   return result;
```

defined by Alg. 2 and, given a regex $\mathtt{r}$ and $i, j \in \mathbb{N}$, it returns a set of triples of the form (σ, n_1, n_2), such that σ is the *partial substring* that Alg. 2 has computed up to now, $n_1 \in \mathbb{N}$ tracks how many characters have still to be skipped before the substring can be computed and $n_2 \in \mathbb{N}$ is the number of characters Alg. 2 needs still to look for to successfully compute a substring. Hence, given $\mathsf{Sb}(\mathtt{r}, i, j)$, the result is a set of such triples; note that given an element of the resulting set (σ, n_1, n_2), $n_2 = 0$ means that no more characters are needed and σ corresponds to a proper substring of $\mathtt{r}$ from i to j. Thus, from the resulting set, we can filter out the partial substrings, and retrieve only proper substrings of $\mathtt{r}$ from i to j, by only considering the value of n_2. Alg. 2 is defined by case on the structure of the input regex $\mathtt{r}$:

1. $j = 0$ or $\mathtt{r} = \varnothing$ (lines 1-2): $\varnothing$ is returned since we either completed the substring or we have no more characters to add;
2. $\mathtt{r} = \sigma \in \Sigma^*$ (lines 3-6): if $i > |\sigma|$, the requested substring happens after this atom, and we return a singleton set $\{\epsilon, i - |\sigma|, j\}$, thus tracking the consumed characters before the start of the requested substring; if $i+j > |\sigma|$, the substring begins in σ but ends in subsequent regexes, and we return a

singleton set containing the substring of σ from i to its end, with $n_1 = 0$ since we begun collecting characters, and $n_2 = j - |\sigma| + i$ since we collected $|\sigma| - i$ characters; otherwise, the substring is fully inside σ, and we return the substring of σ from i to $i + j$, setting both n_1 and n_2 to 0;

3. $\mathtt{r} = \mathsf{T}$ (lines 7-10): since $\mathtt{r}$ might have any length, we generate substrings that (a) gradually consume all the missing characters before the substring can begin (line 8) and (b) gradually consume all the characters that make up the substring, adding the unknown character $\bullet$ (line 9);
4. $\mathtt{r} = \mathtt{r}_1\mathtt{r}_2$ (lines 11-20): the desired substring can either be fully found in $\mathtt{r}_1$ or $\mathtt{r}_2$, or could overlap them; thus we compute all the partial substrings of $\mathtt{r}_1$, recursively calling Sb (line 13); for all $\{\sigma_1, i_1, j_1\}$ returned, substrings that are fully contained in $\mathtt{r}_1$ (i.e., when $j_1 = 0$) are added to the result (line 16) while the remaining ones are joined with ones computed by recursively calling Sb on $\mathtt{r}_2$ with $n_1 = j_1$ and $n_2 = j_2$;
5. $\mathtt{r} = \mathtt{r}_1||\mathtt{r}_2$ (lines 20-21): we return the partial substring of $\mathtt{r}_1$ and the ones of $\mathtt{r}_2$, recursively calling Sb on both of them;
6. $\mathtt{r} = (\mathtt{r}_1)^*$ (lines 22-31): we construct the set of substrings through fixpoint iteration, starting by generating $\{\epsilon, i, j\}$ (corresponding to $\mathtt{r}_1$ repeated 0 times - line 23) and then, at each iteration, by joining all the partial results obtained until now with the ones generated by a further recursive call to Sb, keeping only the joined results that are new (lines 24-30).

Above, we have defined the abstract semantics of `substring` when intervals are constant. When $\wr \mathtt{a}_1 \wr \mathbb{m}^\sharp = [i, j]$ and $\wr \mathtt{a}_2 \wr \mathbb{m}^\sharp = [l, k]$, with $i, j, l, k \in \mathbb{Z}$, the abstract semantics of `substring` is

$$\wr \mathtt{substr}(\mathsf{s}, \mathtt{a}_1, \mathtt{a}_2) \wr \mathbb{m}^\sharp \triangleq \bigsqcup_{a \in [i,j], b \in [l,k], a \leq b} \bigsqcup \mathsf{Min}(\{\ \sigma \mid (\sigma, 0, 0) \in \mathsf{Sb}(\mathtt{r}, a, b - a)\ \})$$

We do not precisely handle the cases when the intervals are unbounded (e.g., $[1, +\infty]$). These cases have been already considered in [8] and treated in an ad-hoc manner and one may recast the same proposed idea in our context. Nevertheless, when these cases are met, our analysis returns the automaton recognizing any possible substring of the input automaton, still guaranteeing soundness.

5 Experimental Results

Tarsis has been compared with five other domains, namely the prefix (Pr), suffix (Su), char inclusion (Ci), bricks (Br) domains (all defined in [16]), and $\text{Fa}_{/\equiv}$ (defined in [8], adapting their abstract semantics definition for Java, without altering their precision).

All domains have been implemented in a prototype of a static analyzer for a subset of the Java language, similar to Imp (Sect. 3), plus the `assert` statement. In particular, our analyzer raises a *definite* alarm (DA for short) when a failing assert (i.e., whose condition is definitely false) is met, while it raises a *possible* alarm (PA for short) when the assertion *might* fail (i.e., the assertion's condition

```
1  void substring() {
2    String res = "substring test";
3    if (nondet)
4      res = res + " passed";
5    else
6      res = res + " failed";
7    result = res.substring(5, 18);
8    assert (res.contains("g"));
9    assert (res.contains("p"));
10   assert (res.contains("f"));
11   assert (res.contains("d"));
12 }
```

(a) Program SUBS

```
1  void loop() {
2    String value = read();
3    String res = "Repeat: ";
4      while (nondet)
5        res = res + value + "!";
6      assert (res.contains("t"));
7      assert (res.contains("!"));
8      assert (res.contains("f"));
9  }
```

(b) Program LOOP

Fig. 7: Program samples used for domain comparison

evaluates to $\mathsf{T}_{\mathsf{Bool}}$). Comparisons have been performed by analyzing the code through the coalesced sum domain specified in Sect. 4.2 with trace partitioning [31] (note that all traces are merged when evaluating an assertion), plugging in the various string domains. All experiments have been performed on a HP EliteBook G6 machine, with an Intel Core i7-8565U @ 1.8GHz processor and 16 GB of RAM memory.

To achieve a fair comparison with the other string domains, the subjects of our evaluation are small hand-crafted code fragments that represent standard string manipulations that occur regularly in software. PR, SU, CI and BR have been built to model simple properties and to work with integers instead of intervals, and have been evaluated on small programs: Sect. 5.1 compares them to TARSIS and $\text{FA}_{/\equiv}$ without expanding the scope of such evaluations. Sect. 5.2 instead focuses on slightly more advanced and complex string manipulations that are not modeled by the aforementioned domains, but that $\text{FA}_{/\equiv}$ and TARSIS can indeed tackle, highlighting differences between them.

It is important to notice that performances of programs relying on automata (highlighted in Sect. 5.3) are heavily dependent on their implementation. Both $\text{FA}_{/\equiv}$ and TARSIS (whose sources are available on GitHub[8,9]) come as non-optimized proof-of-concept libraries (specifically, TARSIS has been built following the structure of $\text{FA}_{/\equiv}$ to ensure a fair performance comparison) whose performances can be greatly improved.

5.1 Precision of the various domains on test cases

We start by considering programs SUBS (Fig. 7a) and LOOP (Fig. 7b). SUBS calls substring on the concatenation between two strings, where the first is constant and the second one is chosen in a non-deterministic way (i.e., nondet condition is statically unknown, lines 3-6). LOOP builds a string by repeatedly appending a suffix, which contains a user input (i.e., an unknown string), to a constant value. Tab. 1 reports the value approximation for res for each abstract domain and analyzed program when the first assertion of each program is met, as well

[8] $\text{FA}_{/\equiv}$ source code: https://github.com/SPY-Lab/fsa
[9] TARSIS source code: https://github.com/UniVE-SSV/tarsis

as if the abstract domain precisely dealt with the program assertions. For the sake of readability, Tarsis and $\text{Fa}_{/\equiv}$ approximations are expressed as regexes.

When analyzing subs, both Pr and Su lose precision since the string to append to res is statically unknown. This leads, at line 7, to a partial substring of the concrete one with Pr, and to an empty string with Su. Instead, the substring semantics of Ci moves every character of the receiver in the set of possibly contained ones, thus the abstract value at line 7 is composed by an empty set of included characters, and a set of possibly included characters containing the ones of both strings. Finally, Br, $\text{Fa}_{/\equiv}$ and Tarsis are expressive enough to track any string produced by any concrete execution of subs.

When evaluating the assertions of subs, a PA should be raised on lines 9 and 10, since "p" or "f" might be in res, together with a DA alarm on line 11, since "d" is surely not contained in res. No alarm should be raised on line 8 instead, since "g" is part of the common prefix of both branches and thus will be included in the substring. Such behavior is achieved when using Br, $\text{Fa}_{/\equiv}$, or Tarsis. Since the substring semantics of Ci moves all characters to the set of possibly contained ones, PAs are raised on all four assertions. Since Su loses all information about res, PAs are raised on lines 7-10 when using such domain. Pr instead tracks the definite prefix of res, thus the PA at line 7 is avoided.

When analyzing loop, we expect to obtain no alarm at line 6 (since character "t" is always contained in the resulting string value), and PA at lines 7 and 8. Pr infers as prefix of res the string "Repeat: ", keeping such value for the whole analysis of the program. This allows the analyzer to prove the assertion at line 6, but it raises PAs when it checks the ones at lines 7 and 8. Again, Su loses any information about res since the lub operation occurring at line 3 cannot find a common suffix between "Repeat: " and "!", hence PAs are raised on lines 6-8. Since the set of possible characters contains T, Ci can correctly state that any character might appear in the string. For this reason, two PAs are reported on lines 7 and 8, while no alarm is raised on line 6 (again, this is possible since the string used in the contains call has length 1). The alternation of T and "!" prevents Br normalization algorithm from merging similar bricks. This will eventually lead to overcoming the length threshold k_L, hence resulting in the $[\{\mathsf{T}\}]\,(0, +\infty)$ abstract value. In such a situation, Br returns $\mathsf{T}_{\mathsf{Bool}}$ on all contains calls, resulting in PAs on lines 6-8. The parametric widening of $\text{Fa}_{/\equiv}$ collapses the colon into T. In Tarsis, since the automaton representing res grows by two states each iteration, the parametric widening defined in Sect. 4.1 can collapse the whole content of the loop into a 2-states loop recognizing T!. The

Domain	**Program** subs		**Program** loop	
Pr	ring test	✗	Repeat:	✗
Su	ϵ	✗	ϵ	✗
Ci	[] [abdefgilnprstu]	✗	[:aepRt] [!:aepRt T]	✓
Br	$[\{\text{ring test fai}, \text{ring test pas}\}]\,(1,1)$	✓	$[\{\mathsf{T}\}]\,(0, +\infty)$	✗
$\text{Fa}_{/\equiv}$	ring test (pas||fai)	✓	Repeat: $(\mathsf{T})^*$	✓
Tarsis	(ring test pas||ring test fai)	✓	Repeat: $(\mathsf{T}!)^*$	✓

Table 1: Values of res at the first assert of each program

```
1  void toString(String[] names) {
2    String res="People: {";
3    int i=0;
4    while(i<names.length){
5      res=res+names[i];
6      if(i!=names.length-1)
7        res=res+",";
8      i=i+1;
9    }
10   res=res+"}";
11   assert(res.contains("People"));
12   assert(res.contains(","));
13   assert(res.contains("not"));
14 }
```

(a) Program toString

```
1  void count(boolean nondet) {
2    String str;
3    if(nondet) str="this is the thing";
4    else str="the throat";
5    int count=countMatches(str, "th")
6    assert(count>0);
7    assert(count==0);
8    assert(count==3);
9  }
```

(b) Program count

Fig. 8: Programs used for assessing domain precision

precise approximation of res of both domains enable the analyzer to detect that the assertion at line 6 always holds, while PAs are raised on lines 7 and 8.

In summary, Pr and Su failed to produce the expected results on both subs and loop, while Ci and Br produced exact results in one case (loop and subs, respectively), but not in the other. Hence, $\text{Fa}_{/\equiv}$ and Tarsis were the two only domains that produced the desired behavior in these rather simple test cases.

5.2 Evaluation on realistic code samples

In this section, we explore two real world code samples. Method toString (Fig. 8a) transforms an array of names that come as string values into a single string. While it resembles the code of loop in Fig. 7b (thus, results of all the analyses show the same strengths and weaknesses), now assertions check contains predicates with a multi-character string. Method count (Fig. 8b) makes use of countMatches (reported in Sect. 2) to prove properties about its return value. Since the analyzer is not inter-procedural, we inlined countMatches inside count. Tab. 2 reports the results of both methods (stored in res and count, respectively) evaluated by each analysis at the first assertion, as well as if the abstract domain precisely dealt with the program assertions.

As expected, when analyzing toString, each domain showed results similar to those of loop. In particular, we expect to obtain no alarm at line 11 (since "People" is surely contained in the resulting string), and two PAs at line 12 and 13. Pr, Su, Ci and Br raise PAs on all the three assert statements. $\text{Fa}_{/\equiv}$ and

Domain	Program toString		Program count	
Pr	People: {	✗	$[0, +\infty]$	✗
Su	ϵ	✗	$[0, +\infty]$	✗
Ci	[{}:Peopl] [{}:,Peopl T]	✗	$[0, +\infty]$	✗
Br	[{T}] $(0, +\infty)$	✗	$[0, +\infty]$	✗
$\text{Fa}_{/\equiv}$	People: {(T)*T}	✓	$[2, 3]$	✓
Tarsis	People: {}||People: {(T,)*T}	✓	$[2, 3]$	✓

Table 2: Values of res and count at the first assert of the respective program

Domain	SUBS	LOOP	TOSTRING	COUNT
PR	11 ms	3 ms	78 ms	29 ms
SU	10 ms	2 ms	92 ms	29 ms
CI	10 ms	3 ms	90 ms	29 ms
BR	13 ms	3 ms	190 ms	28 ms
$\text{FA}_{/\equiv}$	10 ms	52013 ms	226769 ms	4235 ms
TARSIS	34 ms	38 ms	299 ms	39 ms

Table 3: Execution times of the domains on each program

TARSIS detect that the assertion at line 11 always holds. Thus, when using them, the analyzer raises PAs on lines 12 and 13 since: comma character is part of res if the loop is iterated at least once, and T might match "not".

If COUNT (with the inlined code from COUNTMATCHES) was to be executed, count would be either 2 or 3 when the first assertion is reached, depending on the choice of str. Thus, no alarm should be raised at line 6, while a DA should be raised on line 7, and a PA on line 8. Since PR, SU, CI and BR do not define most of the operations used in the code, the analyzer does not have information about the string on which COUNTMATCHES is executed, and thus abstract count with the interval $[0, +\infty]$. Thus, PAs are raised on lines 6-8. Instead, $\text{FA}_{/\equiv}$ and TARSIS are instead able to detect that sub is present in all the possible strings represented by str. Thus, thanks to trace partitioning, the trace where the loop is skipped and count remains 0 gets discarded. Then, when the first indexOf call happens, $[0, 0]$ is stored into idx, since all possible values of str start with sub. Since the call to length yields $[10, 17]$, all possible substrings from $[2, 2]$ (idx plus the length of sub) to $[10, 17]$ are computed (namely, "e throat", "is is th", "is is the", ..., "is is the thing"), and the resulting automaton is the one that recognizes all of them. Since the value of sub is still contained in every path of such automaton, the loop guard still holds and the second iteration is analyzed, repeating the same operations. When the loop guard is reached for the third time, the remaining substring of the shorter starting string (namely "roat") recognized by the automaton representing str will no longer contain sub: a trace where count equals $[2, 2]$ will leave the loop. A further iteration is then analyzed, after which sub is no longer contained in any of the strings that str might hold. Thus, a second and final trace where count equals $[3, 3]$ will reach the assertions, and will be merged by interval lub, obtaining $[2, 3]$ as final value for count. This allows TARSIS and $\text{FA}_{/\equiv}$ to identify that the assertion at line 7 never holds, raising a DA, while the one at line 8 might not hold, raising a PA.

5.3 Efficiency

The detailed analysis of two test cases, and two examples taken from real-world code underlined that TARSIS and $\text{FA}_{/\equiv}$ are the only ones able to obtain precise results on them. We now discuss the efficiency of the analyses. Tab. 3 reports the execution times for all the domains on the case studies analyzed in this section. Overall, PR, SU, CI, and BR are the fastest domains with execution times usually below 100 msecs. Thus, if on the one hand these domains failed to prove some

of the properties of interest, they are quite efficient and they might be helpful to prove simple properties. Tarsis execution times are higher but still comparable with them (about 50% overhead on average). Instead, $\text{Fa}_{/\equiv}$ blows up on three out of the four test cases (and in particular on toString). Hence, Tarsis is the only domain that executes the analysis in a limited time while being able to prove all the properties of interest on these four case studies.

The reason behind the performance gap between Tarsis and $\text{Fa}_{/\equiv}$ can be accounted on the alphabets underlying the automata. In $\text{Fa}_{/\equiv}$, automata are built over an alphabet of single characters. While this simplifies the semantic operations, it also causes state and transition blow up w.r.t. the size of the string that needs to be represented. This does not happen in Tarsis, since atomic strings (not built through concatenation or other string manipulations) are part of the alphabet and can be used as transition symbol. Having less states and transitions to operate upon drastically lowers the time and memory requirements of automata operations, making Tarsis faster than $\text{Fa}_{/\equiv}$.

Tarsis's alphabet has another peculiarity w.r.t. $\text{Fa}_{/\equiv}$'s: it has a special symbol for representing the unknown string. Having such a symbol requires some fine-tuning of the algorithms to have them behave differently when the symbol is encountered, but without additional tolls on their performances. $\text{Fa}_{/\equiv}$'s alphabet does not have such a symbol, thus representing the unknown string is achieved through a state having one self-loop for each character in the alphabet (including the empty string). This requires significantly more resources for automata algorithms, leading to higher execution times.

6 Conclusion

In this paper we introduced Tarsis, an abstract domain for sound abstraction of string values. Tarsis is based on finite state automata paired with their equivalent regular expression: a representation that allows precise modeling of complex string values. Experiments show that Tarsis achieves great precision also on code that heavily manipulate string values, while the time needed for the analysis is comparable with the one of other simpler domains.

The analysis proposed in this paper is intra-procedural and we are currently working on extending it to an inter-procedural analysis. Moreover, in order to further improve the performance of our analysis, sophisticated techniques such as abstract slicing [26, 27] can be integrated to keep the size of automata arising during abstract computations as low as possible, by focusing the analysis only on the string variables of interest. Finally, in this paper, we did not investigate completeness property of Tarsis w.r.t. the considered operations of interest. This would ensure that no loss of information is related to $\mathcal{T}\text{Fa}_{/\equiv}$ due to the input abstraction process [9]. Our future directions will include a deeper study about $\mathcal{T}\text{Fa}_{/\equiv}$ completeness, and possibly the application of completion processes when incompleteness arises for a string operation [24].

References

1. Abdulla, P.A., Atig, M.F., Chen, Y., Diep, B.P., Dolby, J., Janku, P., Lin, H., Holík, L., Wu, W.: Efficient handling of string-number conversion. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020. pp. 943–957. ACM (2020). https://doi.org/10.1145/3385412.3386034
2. Abdulla, P.A., Atig, M.F., Chen, Y., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: String constraints for verification. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8559, pp. 150–166. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_10
3. Abdulla, P.A., Atig, M.F., Diep, B.P., Holík, L., Janku, P.: Chain-free string constraints. In: Chen, Y., Cheng, C., Esparza, J. (eds.) Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11781, pp. 277–293. Springer (2019). https://doi.org/10.1007/978-3-030-31784-3_16
4. Almashfi, N., Lu, L.: Precise string domain for analyzing javascript arrays and objects. In: 2020 3rd International Conference on Information and Computer Technologies (ICICT). pp. 17–23 (2020)
5. Amadini, R., Gange, G., Stuckey, P.J.: Dashed strings for string constraint solving. Artificial Intelligence **289**, 103368 (2020). https://doi.org/https://doi.org/10.1016/j.artint.2020.103368
6. Arceri, V., Maffeis, S.: Abstract domains for type juggling. Electron. Notes Theor. Comput. Sci. **331**, 41–55 (2017). https://doi.org/10.1016/j.entcs.2017.02.003
7. Arceri, V., Mastroeni, I.: A sound abstract interpreter for dynamic code. In: Hung, C., Cerný, T., Shin, D., Bechini, A. (eds.) SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing, online event, [Brno, Czech Republic], March 30 - April 3, 2020. pp. 1979–1988. ACM (2020). https://doi.org/10.1145/3341105.3373964
8. Arceri, V., Mastroeni, I., Xu, S.: Static analysis for ecmascript string manipulation programs. Appl. Sci. **10**, 3525 (2020). https://doi.org/10.3390/app10103525
9. Arceri, V., Olliaro, M., Cortesi, A., Mastroeni, I.: Completeness of abstract domains for string analysis of javascript programs. In: Hierons, R.M., Mosbah, M. (eds.) Theoretical Aspects of Computing - ICTAC 2019 - 16th International Colloquium, Hammamet, Tunisia, October 31 - November 4, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11884, pp. 255–272. Springer (2019). https://doi.org/10.1007/978-3-030-32505-3_15
10. Bartzis, C., Bultan, T.: Widening arithmetic automata. In: Alur, R., Peled, D.A. (eds.) Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3114, pp. 321–333. Springer (2004). https://doi.org/10.1007/978-3-540-27813-9_25
11. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. Proc. ACM Program. Lang. **3**(POPL), 49:1–49:30 (2019). https://doi.org/10.1145/3290362

12. Choi, T., Lee, O., Kim, H., Doh, K.: A practical string analyzer by the widening approach. In: Kobayashi, N. (ed.) Programming Languages and Systems, 4th Asian Symposium, APLAS 2006, Sydney, Australia, November 8-10, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4279, pp. 374–388. Springer (2006). https://doi.org/10.1007/11924661_23
13. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: Cousot, R. (ed.) Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2694, pp. 1–18. Springer (2003). https://doi.org/10.1007/3-540-44898-5_1
14. Cortesi, A., Olliaro, M.: M-string segmentation: A refined abstract domain for string analysis in C programs. In: Pang, J., Zhang, C., He, J., Weng, J. (eds.) 2018 International Symposium on Theoretical Aspects of Software Engineering, TASE 2018, Guangzhou, China, August 29-31, 2018. pp. 1–8. IEEE Computer Society (2018). https://doi.org/10.1109/TASE.2018.00009
15. Cortesi, A., Zanioli, M.: Widening and narrowing operators for abstract interpretation. Comput. Lang. Syst. Struct. **37**(1), 24–42 (2011). https://doi.org/10.1016/j.cl.2010.09.001
16. Costantini, G., Ferrara, P., Cortesi, A.: A suite of abstract domains for static analysis of string values. Softw. Pract. Exp. **45**(2), 245–287 (2015). https://doi.org/10.1002/spe.2218
17. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977. pp. 238–252. ACM (1977). https://doi.org/10.1145/512950.512973
18. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Aho, A.V., Zilles, S.N., Rosen, B.K. (eds.) Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979. pp. 269–282. ACM Press (1979). https://doi.org/10.1145/567752.567778
19. Cousot, P., Cousot, R.: Abstract interpretation frameworks. J. Log. Comput. **2**(4), 511–547 (1992). https://doi.org/10.1093/logcom/2.4.511
20. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Aho, A.V., Zilles, S.N., Szymanski, T.G. (eds.) Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978. pp. 84–96. ACM Press (1978). https://doi.org/10.1145/512760.512770
21. D'Antoni, L., Veanes, M.: Minimization of symbolic automata. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014. pp. 541–554. ACM (2014). https://doi.org/10.1145/2535838.2535849
22. Davis, M.D., Sigal, R., Weyuker, E.J.: Computability, Complexity, and Languages: Fund. of Theor. CS. Academic Press Professional, Inc. (1994)
23. D'Silva, V.: Widening for Automata. MsC Thesis, Inst. Fur Inform. - UZH (2006)
24. Giacobazzi, R., Ranzato, F., Scozzari, F.: Making abstract interpretations complete. J. ACM **47**(2), 361–416 (2000). https://doi.org/10.1145/333979.333989, https://doi.org/10.1145/333979.333989
25. Madsen, M., Andreasen, E.: String analysis for dynamic field access. In: Cohen, A. (ed.) Compiler Construction - 23rd International Conference, CC 2014,

Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8409, pp. 197–217. Springer (2014). https://doi.org/10.1007/978-3-642-54807-9_12

26. Mastroeni, I., Nikolic, D.: Abstract program slicing: From theory towards an implementation. In: Dong, J.S., Zhu, H. (eds.) Formal Methods and Software Engineering - 12th International Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China, November 17-19, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6447, pp. 452–467. Springer (2010). https://doi.org/10.1007/978-3-642-16901-4_30
27. Mastroeni, I., Zanardini, D.: Abstract program slicing: An abstract interpretation-based approach to program slicing. ACM Trans. Comput. Log. **18**(1), 7:1–7:58 (2017). https://doi.org/10.1145/3029052, https://doi.org/10.1145/3029052
28. Midtgaard, J., Nielson, F., Nielson, H.R.: A parametric abstract domain for lattice-valued regular expressions. In: Rival, X. (ed.) Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9837, pp. 338–360. Springer (2016). https://doi.org/10.1007/978-3-662-53413-7_17
29. Park, C., Im, H., Ryu, S.: Precise and scalable static analysis of jquery using a regular expression domain. In: Ierusalimschy, R. (ed.) Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Amsterdam, The Netherlands, November 1, 2016. pp. 25–36. ACM (2016). https://doi.org/10.1145/2989225.2989228
30. Preda, M.D., Giacobazzi, R., Lakhotia, A., Mastroeni, I.: Abstract symbolic automata: Mixed syntactic/semantic similarity analysis of executables. In: Rajamani, S.K., Walker, D. (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015. pp. 329–341. ACM (2015). https://doi.org/10.1145/2676726.2676986
31. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. ACM Trans. Program. Lang. Syst. **29**(5), 26–es (Aug 2007). https://doi.org/10.1145/1275497.1275501, https://doi.org/10.1145/1275497.1275501
32. Veanes, M.: Applications of symbolic finite automata. In: Konstantinidis, S. (ed.) Implementation and Application of Automata - 18th International Conference, CIAA 2013, Halifax, NS, Canada, July 16-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7982, pp. 16–23. Springer (2013). https://doi.org/10.1007/978-3-642-39274-0_3
33. Wang, H., Chen, S., Yu, F., Jiang, J.R.: A symbolic model checking approach to the analysis of string and length constraints. In: Huchard, M., Kästner, C., Fraser, G. (eds.) Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018. pp. 623–633. ACM (2018). https://doi.org/10.1145/3238147.3238189
34. Yu, F., Alkhalaf, M., Bultan, T., Ibarra, O.H.: Automata-based symbolic string analysis for vulnerability detection. Formal Methods Syst. Des. **44**(1), 44–70 (2014). https://doi.org/10.1007/s10703-013-0189-1