Autom Softw Eng (2012) 19:531–559
DOI 10.1007/s10515-012-0111-x

STRSOLVE: solving string constraints lazily

Pieter Hooimeijer · Westley Weimer

Received: 26 August 2011 / Accepted: 28 May 2012 / Published online: 3 July 2012
© Springer Science+Business Media, LLC 2012

Abstract Reasoning about strings is becoming a key step at the heart of many program analysis and testing frameworks. Stand-alone string constraint solving tools, called decision procedures, have been the focus of recent research in this area. The aim of this work is to provide algorithms and implementations that can be used by a variety of program analyses through a well-defined interface. This separation enables independent improvement of string constraint solving algorithms and reduces client effort.

We present STRSOLVE, a decision procedure that reasons about equations over string variables. Our approach scales well with respect to the size of the input constraints, especially compared to other contemporary techniques. Our approach performs an explicit search for a satisfying assignment, but constructs the search space lazily based on an automata representation. We empirically evaluate our approach by comparing it with four existing string decision procedures on a number of tasks. We find that our prototype is, on average, several orders of magnitude faster than the fastest existing approaches, and present evidence that our lazy search space enumeration accounts for most of that benefit.

Keywords String · Regular language · Decision procedure · Scalability

We gratefully acknowledge the support of the National Science Foundation (grants CCF-0905236, CCF-0954024 and CNS-0716478), Air Force Office of Scientific Research grant FA8750-11-2-0039, MURI grant FA9550-07-1-0532, and DARPA grant FA8650-10-C-7089.

P. Hooimeijer (✉) · W. Weimer
85 Engineer's Way, P.O. Box 400740, Charlottesville, VA 22904-4740, USA
e-mail: pieter@cs.virginia.edu

W. Weimer
e-mail: weimer@cs.virginia.edu

1 Introduction

Reasoning about string variables is a key aspect in many areas of program analysis, including the analysis of dynamically-generated content (Christensen et al. 2003), web pages (Minamide 2005), injection vulnerabilities (Su and Wassermann 2006; Wassermann and Su 2007), and scripting languages (Xie and Aiken 2006). It is also prevalent in the domain of automated test input generation (Godefroid et al. 2005), both white-box (Godefroid et al. 2008b) and grammar-based (Godefroid et al. 2008a; Majumdar and Xu 2007). Program analyses and transformations that deal with string-manipulating programs, such as test input generation for legacy systems (Lakhotia et al. 2008, 2009), web application bug finding (Wassermann and Su 2007), and program repair (Weimer et al. 2009), invariably require a model of string manipulating functions.

Traditionally, both static and dynamic analyses have relied on their own built-in models to reason about constraints on string variables, just as early analyses relied on built-in conservative reasoning about aliasing. The current situation is suboptimal for two reasons: first, it forces researchers to re-invent the wheel for each new tool; and second, it inhibits the independent improvement of algorithms for reasoning about strings.

External constraint solving tools have long been available for other domains, such as satisfiability modulo theories (SMT) (de Moura and Bjørner 2008; Detlefs et al. 2005; Necula 1997) and boolean satisfiability (SAT) (Eén and Sörensson 2003; Moskewicz et al. 2001; Xie and Aiken 2005). Recent work in string analysis has focused on providing similar external decision procedures for string constraints (Axelsson et al. 2008; Hooimeijer and Weimer 2009; Kiezun et al. 2009; Veanes et al. 2010; Yu et al. 2009b, 2010; Hooimeijer and Veanes 2011). Thus far, this work has focused on adding features such as support for symbolic integer constraints (Yu et al. 2009b, 2010), support for bounded context-free grammars (Axelsson et al. 2008; Kiezun et al. 2009), and embedding into an existing SMT solver (de Moura and Bjørner 2008). We argue that the existing approaches leave significant room for improvement with regard to scalability.

We propose a novel decision procedure that supports the efficient, lazy processing of string constraints without requiring a priori length bounds. Our approach is based on the insight that many existing solvers do more work than is strictly necessary because they eagerly encode the search space of possible solutions before searching it. For example, the Hampi tool (Kiezun et al. 2009) performs an eager bitvector encoding of all positional shifts for each regular expression in the given constraint system. We observe that much of that encoding work is unnecessary if the goal is to find a single string assignment as quickly as possible.

Our approach uses an automaton-based representation of string constraint systems. In contrast with previous automaton-based approaches (Hooimeijer and Weimer 2009; Balzarotti et al. 2008; Yu et al. 2009b, 2010), our approach separates the description of the search space from its (potentially expensive) instantiation. For example, when intersecting two automata using the cross-product construction (Sipser 1997), we generate only those parts of the intersection automaton needed to find a single string. Our search space consists of sets of nodes in lazily-constructed finite automata corresponding to string variables and constrained by string operations.

The primary contributions of this paper are:

- A novel decision procedure that supports the efficient and lazy analysis of string constraints. We treat string constraint solving as an explicit search problem, and separate the description of the search space from the search strategy used to traverse it.
- A comprehensive performance comparison between our prototype tool and existing implementations. We compare against CFG Analyzer (Axelsson et al. 2008), DPRLE (Hooimeijer and Weimer 2009), and Hampi (Kiezun et al. 2009). We use several sets of established benchmarks (Kiezun et al. 2009; Veanes et al. 2010). We find that our prototype is several orders of magnitude faster for the majority of benchmark inputs; for all other inputs our performance is, at worst, competitive with existing methods.

Some of these main points were previously presented (Hooimeijer and Weimer 2010). Since then, we have made our source code and implementation publicly available.[1] This article also includes an extended presentation of the main algorithm and related work, in addition to the follow points:

- A new experiment focusing on the performance of the Hampi (Kiezun et al. 2009) tool. The first three experiments we present all support the claim that lazy search space enumeration is the key to our algorithm's performance. For example, the new experiment demonstrates that even if Hampi were to replace its underlying solver with a zero-time oracle, it would still under-perform our approach because of its exponential preprocessing and encoding step.
- An explicit comparison to the Rex tool (Veanes et al. 2010) for two experiments: regular set difference and generating long strings. While Rex exhibits better asymptotic scaling behavior than DPRLE or Hampi, it is still at least an order of magnitude slower than our prototype.
- A worked example highlighting how our approach handles cyclic constraints in Sect. 2. The example was chosen based on feedback received after earlier publications, and details how our backtracking search strategy gracefully handles constraint systems with a cyclic order dependency across two variables. Such systems arise, for example, in string manipulating programs that reference both "*a* concatenated with *b*" and also "*b* concatenated with *a*".
- Additional detail about the core `advance` algorithm that guides the lazy search (Sect. 2.5), and a more elaborate proof sketch based on that presentation (Sect. 2.6).

The structure of this paper is as follows. In Sect. 2, we provide a high-level overview of our algorithm, focusing on the (eager) construction of a graph-based representation of the search space (Sect. 2.2), followed by the (lazy) traversal of the search space (Sect. 2.3). We provide two worked examples of the algorithm in Sect. 2.4, additional detail about the intersection algorithm in Sect. 2.5, and an informal correctness argument in Sect. 2.6. Section 3 provides performance results, focusing on regular language difference (Sect. 3.1), the detailed performance characteristics of an existing approach (Sect. 3.2), regular intersection for large strings

[1] http://code.google.com/p/strsolve/.

(Sect. 3.3), and bounded context-free intersection (Sect. 3.4). Section 4 elaborates on related work, and we conclude in Sect. 5.

2 Approach

In the following subsections, we present our decision procedure for string constraints. Our goal is to provide expressiveness similar to that of existing tools such as DPRLE, Rex, and Hampi (Hooimeijer and Weimer 2009; Veanes et al. 2010; Kiezun et al. 2009), while exhibiting significantly improved average-case performance. In Sect. 2.1, we formally define the string constraints of interest. Section 2.2 outlines our high-level graph representation of problem instances. We then provide an algorithm for finding satisfying assignments in Sect. 2.3, and work through illustrative examples in Sect. 2.4. Finally, we provide additional algorithmic detail in Sect. 2.5, and a correctness proof sketch in Sect. 2.6.

2.1 Definitions

In this work, we focus on a set of string constraints over regular languages similar to the regular constraints presented by Kiezun et al. (2009), but without requiring a priori bounds on string variable length. In earlier work (Hooimeijer and Weimer 2009), we demonstrate that this type of string constraint can model a variety of common programming language constructs.

The set of well-formed string constraints is defined by the grammar in Fig. 1. A constraint system S is a set of constraints of the form $S = \{C_1, \ldots, C_n\}$, where each $C_i \in S$ is derivable from *Constraint* in Fig. 1. *Var* denotes a finite set of string variables $\{v_1, \ldots, v_m\}$. *ConstVal* denotes the set of string literals. For example, $v \in \texttt{ab}$ denotes that variable v must have the constant value `ab` for any satisfying assignment. We describe inclusion and non-inclusion constraints symmetrically when possible, using $\diamond$ to represent either relation (i.e., $\diamond \in \{\in, \notin\}$).

Constraint	::=	*StringExpr* $\in$ *RegExpr*	inclusion
	|	*StringExpr* $\notin$ *RegExpr*	non-inclusion
StringExpr	::=	*Var*	string variable
	|	*StringExpr* $\circ$ *Var*	concat
RegExpr	::=	*ConstVal*	string literal
	|	*RegExpr* + *RegExpr*	language union
	|	*RegExpr RegExpr*	language concat
	|	*RegExpr**	Kleene star

Fig. 1 String inclusion constraints for regular sets. A constraint system is a set of constraints over a shared set of string variables; a satisfying assignment maps each string variable to a value so that all constraints are simultaneously satisfied. *ConstVal* represents a string literal; *Var* represents an element in a finite set of shared string variables

For a given constraint system S over variables $\{v_1, \ldots, v_m\}$, we write $A = [v_1 \leftarrow x_1, \ldots, v_m \leftarrow x_m]$ for the *assignment* that maps variables $v_1, \ldots, v_m$ to values $x_1, \ldots, x_m$, respectively. We define $[\![v_i]\!]_A$ to be the value of v_i under assignment A; for a *StringExpr* E, $[\![E \circ v_i]\!]_A = [\![E]\!]_A \circ [\![v_i]\!]_A$. For a *RegExpr* R, $[\![R]\!]$ denotes the set of strings in the language $L(R)$, following the usual interpretation of regular expressions. When convenient, we equate a regular expression literal like $\texttt{ab}^\star$ with its language. We refer to the negation of a language using a bar (e.g., $\overline{\texttt{ab}^\star} = \{w \mid w \notin \texttt{ab}^\star\}$).

An assignment A for a system S over variables $\{v_1, \ldots, v_m\}$ is *satisfying* iff for each constraint $C_i = E \diamond R$ in the system S, it holds that $[\![E]\!]_A \diamond [\![R]\!]$. We call constraint system S *satisfiable* if there exists at least one satisfying assignment; alternatively we will refer to such a system as a *yes–instance*. A system for which no satisfying assignment exists is *unsatisfiable* and a *no–instance*. A *decision procedure* for string constraints is an algorithm that, given a constraint system S, returns a satisfying assignment for S iff one exists, or "Unsatisfiable" iff no satisfying assignment exists.

We distinguish between a regular expression R and its representation as a nondeterministic finite state automaton, $\mathsf{nfa}(R)$. When discussing pseudocode, we adopt the notation $\mathsf{nfa}(R).q$ when it is necessary to refer to a particular state in $\mathsf{nfa}(R)$ through metavariable q. We use metavariables s and f to refer to the start and final state of an automaton; we assume without loss of generality that automata have a single final state.

2.2 Follow graph construction

We now turn to the problem of efficiently finding satisfying assignments for string constraint systems. We break this problem into two parts. First, in this subsection, we develop a method for eagerly constructing a high-level description of the search space. Then, in Sect. 2.3, we describe a lazy algorithm that uses this high-level description to search the space of satisfying assignments.

For a given constraint system I, we define a *follow graph*, G, as follows:

- For each string variable v_i, the graph has a single corresponding vertex $\mathsf{node}(v_i)$.
- For each occurrence of $\ldots v_i \circ v_j \ldots$ in a constraint in I, the graph has a directed edge from $\mathsf{node}(v_i)$ to $\mathsf{node}(v_j)$. This edge encodes the fact that the satisfying assignment for v_j must immediately follow v_i's.

We also maintain a mapping M from individual constraints in I to their corresponding path through the follow graph. For each constraint $C_h = v_j \diamond R$, we map C_h to path $[\mathsf{node}(v_j)]$. For each constraint C_i of the form $v_k \circ \ldots v_m \diamond R$, we map C_i to path $[\mathsf{node}(v_k), \ldots, \mathsf{node}(v_m)]$.

Figure 2 provides high-level pseudocode for constructing the follow graph for a given system. The follow_graph procedure takes a constraint system I and outputs a pair (G, M), where G is the follow graph corresponding to I, and M is the associated mapping from constraints in I to paths through G. For each constraint in I (line 4), we add edges for each adjacent pair of variables in the constraint (lines 5–7), and update M with the resulting path (line 8). For line 5, we assume that singleton constraints of the form $v_1 \diamond R$ are matched as well; this results in zero edges added (lines 6–7) and a singleton path $[\mathsf{node}(v_1)]$ (line 8).

```
1: follow_graph(I : constraint system) =
2:   let G : directed graph = empty
3:   let M : constraint → path = empty
4:   foreach C_i : constraint ∈ I do
5:     let (v_1 ∘ ... ∘ v_n ◇ R) = C_i
6:     for j ∈ 1, ..., n − 1 do
7:       G ← add_edge(G, node(v_j), node(v_{j+1}))
8:     M[C_i] ← [node(v_1), ..., node(v_n)]
9:   return (G, M)
```

Fig. 2 Follow graph generation. Given a constraint system I, we output follow graph G and mapping M (defined in the text). G and M capture the high-level structure of the search space of assignments. The node function returns a distinct vertex for each variable

As an example, consider the following constraint system and its associated follow graph:

$$C_1 = (v_1 \in \mathtt{a}^\star)$$
$$C_2 = (v_2 \in \mathtt{ab})$$
$$C_3 = (v_1 \circ v_2 \in \mathtt{ab})$$

$\vdash C_1 \dashv$ $\vdash C_2 \dashv$

$n_1 \rightarrow n_2$

$\vdash C_3 \dashv$

We represent the graph G with circular vertices. The C annotations represent the domain of the mapping M. We let $n_i = \mathsf{node}(v_i)$. The first two constraints result in the mapping from C_1 to $[n_1]$ and C_2 to $[n_2]$; the third constraint adds the mapping from C_3 to $[n_1, n_2]$. When convenient, we will use variables in place of their corresponding follow graph nodes.

2.3 Lazy state space exploration

Given a follow graph G, and a constraint-to-path mapping M, our goal is to determine whether the associated constraint system has a satisfying assignment. We treat this as a search problem; the search space consists of possible mappings from variables to paths through finite automata (NFAs). We find this variables-to-NFA-paths mapping through a backtracking depth-first search. If the search is successful, then we extract a satisfying assignment from the search result. If we fail to find a mapping, then it is guaranteed not to exist, and we return "Unsatisfiable." In the remainder of this subsection, we will discuss the search algorithm; we walk through two runs of the algorithm in Sect. 2.4.

The NFAs used throughout the algorithm are generated directly from the regular expressions in the original constraint system; our implementation uses an algorithm similar to one presented by Ilie and Yu (2003). For constraints of the form $\cdots \in R$, we construct an NFA that corresponds to $L(R)$ directly. For constraints of the form $\cdots \notin R$, we eagerly construct an NFA that accepts $L(R)$. We then use a lazy version of the powerset construction to determinize and negate that NFA (e.g., Sipser 1997). For this presentation, we assume without loss that each NFA has a single final state.

2.3.1 The search algorithm

For clarity, we will distinguish between *restrictions* on variables imposed by the algorithm and *constraints* in the input constraint system. Our search starts by considering all variables to be unrestricted. We then iteratively pick one of the variables to restrict; doing this typically imposes further restrictions on other variables as well. The order in which we apply restrictions to variables does not affect the eventual outcome of the algorithm (i.e., "Satisfiable" or "Unsatisfiable"), but it may affect how quickly we find the answer. During the search, if we find that we have over-restricted one of the variables, then we backtrack and attempt a different way to satisfy the same restrictions. At the end of the search, there are two possible scenarios:

- At the end of a successful search, each occurrence of a variable in the original constraint system will be mapped to an NFA path; all paths for a distinct variable will have at least one string in common. We return "Satisfiable" and provide one string for each variable.
- At the end of an unsuccessful search, we have searched all possible NFA path assignments for at least one variable, finding no internally consistent mapping for at least one of those variables. There is no need to explore the rest of the state space, since adding constraints cannot create new solutions. We return "Unsatisfiable."

Figure 3 provides high-level pseudocode for the search algorithm. The main entry point is search (lines 9–19), which returns a result (line 1). An assignment (line 1) is a satisfying assignment that maps each variable to a string. The search procedure performs a depth-first traversal of a (lazily constructed) search space; the stack S (line 12) always holds the current path through the tree. Each vertex in the search tree represents a mapping from string variables to restrictions; each edge represents the application of one or more additional restrictions relative to the source vertex.

Each iteration of the main loop (lines 13–18) consists of a call to visit_state. The visit_state procedure takes the current search state, attempts to advance the search, and returns a stepresult (lines 6–7) signaling success or failure. If visit_state returns Next, then we advance the search by pushing the provided search state onto the stack (line 16). If visit_state returns Back, then we backtrack a single step by popping the current state from the stack (line 17). If visit_state returns Done, then we extract a satisfying string assignment from the paths in current search state (line 18). Finally, if the algorithm is forced to backtrack beyond the initial search state, we return Unsat (line 19).

2.3.2 Manipulating the search state

The searchstate type (line 5) captures the bookkeeping needed to perform the search. The *next* element stores which string variable the algorithm will try to further restrict; once set, this will remain the same for potential subsequent visits to the same search state. The *states* element holds the restrictions for each variable for each occurrence of that variable in the constraint system. For example, in the constraint system

$$C_1 = (v_1 \circ v_1 \in R_1)$$

```
1:  datatype result = Unsat of result | Sat of assignment → result
2:  datatype status = Unknown of status | StartsAt of nfastate → status
3:                  | Path of nfapath → status
4:  datatype pos = (constraint × int)
5:  datatype searchstate = { next : var; states : var → pos → status}
6:  datatype stepresult = Next of searchstate → stepresult
7:                      | Back of stepresult | Done of stepresult
8:
9:  search(followgraph G, mapping M) =
10:   let Q : var → pos → status = start_states(M)
11:   let O : searchstate = { next = nil; states = Q}
12:   let S : searchstate stack = [O]
13:   while S is not empty do
14:     let O_cur : searchstate = top(S)
15:     let R : stepresult = visit_state(O_cur, G, M)
16:     match R with Next(O′) → push(O′, S)
17:                | Back → pop(S)
18:                | Done → return Sat(extract(O_cur))
19:   return Unsat
20:
21: visit_state(searchstate O, followgraph G, mapping M) =
22:   if ∀v : node ∈ G, all_paths(O.states[v]) then
23:     return Done
24:   if O.next = nil then
25:     O.next ← pick_advance(O, G, M)
26:   let (success, restrs) = advance(O, G, M)
27:   if ¬success then
28:     return Back
29:   let O′ : searchstate = copy(O)
30:   O′.next ← nil
31:   O′.states[O.next] ← restrs
32:   foreach n : var ∈ succ(O.next, G) do
33:     foreach p = (C, i) : pos s.t. O′.states[O.next][p] = Path(x) ∧
34:                 O′.states[n][(C, i + 1)] = Unknown do
35:       O′.states[n][(C, i + 1)] ← StartsAt(last(x))
36:   return Next(O′)
```

Fig. 3 Lazy backtracking search algorithm for multivariate string constraints. The search procedure performs an explicit search for satisfying assignments. Each occurrence of a variable in the constraint system is initially unconstrained (Unknown) or constrained to an NFA start state (StartsAt). Each call to visit_state attempts to move one or more occurrences from Unknown to StartsAt or from StartsAt to Path. The goal is to reach a searchstate in which each occurrence is constrained to a concrete Path through an NFA. Other procedures (e.g., start_states, extract, and advance) are described in the text

variable v_1 occurs at *positions* (line 4) $(C_1, 1)$ and $(C_1, 2)$. The searchstate maps each variable at each position to a status (lines 2–3), which represents the current restrictions on that occurrence as follows:

1. Unknown (line 2)—This status indicates that we do not know where the NFA path for this variable occurrence should start. In the example, the $(C_1, 2)$ occurrence of v_1 will initially map to Unknown, since its start state depends on the final state of the v_1 occurrence at $(C_1, 1)$.
2. StartsAt (line 2)—This status indicates that we know at which NFA state we should start looking for an NFA path for this variable occurrence. In the example, the $(C_1, 1)$ occurrence of v_1 will initially map to StartsAt(nfa$(C_1).s$), where nfa$(C_1).s$ denotes the start state of the NFA for regular expression R_1.
3. Path (line 3)—This status indicates that we have restricted the occurrence to a specific path through the NFA for the associated constraint. If a variable has multiple occurrences mapped to Path status, then those paths must agree (i.e., have at least one string in common).

Note that these restrictions are increasingly specific. Each non-backtracking step of the algorithm moves at least one variable occurrence from Unknown to StartsAt or from StartsAt to Path. Conversely, each backtracking step consists of at least one move in the direction Path → StartsAt → Unknown.

The majority of the pseudocode in Fig. 3 deals with the manipulation of searchstate instances. The start_states call (line 10) generates the initial restrictions that start the search; it is defined for each variable v for each valid position (C, i) as follows:

$$\text{start_states}(M)[v]\big[(C,i)\big] = \begin{cases} \text{Unknown} & \textbf{if } i > 1 \\ \text{StartsAt}(\text{nfa}(C).s) & \textbf{if } i = 1 \end{cases}$$

The visit_state procedure advances the search by generating new search states (children in the search tree) based on a given search state (the parent). On lines 22–23, we check to see if all variable occurrences have a Path restriction. The corresponding NFA paths are required to agree by construction. In other words, the algorithm would never reach a search state with all Path restrictions unless the path assignments were internally consistent. We continue if there exists at least one non-Path restriction.

The call to pick_advance determines which variable we will try to restrict in this visit and any subsequent visits to this search state. This function determines the order in which we restrict the variables in the constraint system. The order is irrelevant for correctness as long as pick_advance selects each variable frequently enough to guarantee termination of the search. However, for non-cyclic parts of the follow graph, it is generally beneficial to select predecessor nodes (variables) in the follow graph before their successors. This is because visiting the predecessor can potentially change some of the successor's Unknown restrictions to StartsAt restrictions. We leave a more detailed analysis of search heuristics for future work.

The remainder of visit_state deals with tightening restrictions:

- The call to advance (line 26) moves the restrictions on all occurrences of variable $O.next$, along the trajectory Unknown → StartsAt→ Path. To produce Path restrictions, or rule out that a valid path restriction exists, advance performs lazy simultaneous NFA intersection for all occurrences of $O.next$. We discuss advance in more detail in Sect. 2.5.

– If the call to advance succeeds, then the search state generation code of lines 32–35 uses the additional Path restrictions (if any) for $O.next$ to update $O.next$'s successors in the follow graph (if any; succ(v,G) returns the set of immediate successors of v in G). This step exclusively converts Unknown restrictions to StartsAt restrictions. The intuition here is that, if v_2 follows v_1 in some constraint, then the first state for that occurrence of v_2 must match the last state for v_1; last(x) (line 35) returns the last state in NFA path x.

Note that the first step (the call to advance) can potentially fail if $O.next$ proves to be over-restricted. When this occurs, we backtrack (lines 17 and 28) and return to a previous state, causing that state to be visited a second time. These subsequent visits will lead to repeated call to advance on the same parameters; we assume that advance keeps internal state to ensure that it exhaustively attempts all distinct combinations of StartsAt restrictions. For now, we leave advance abstract and provide worked examples that focus on the execution of the main search algorithm (Sect. 2.4). In Sect. 2.5, we describe advance in more detail.

2.4 Worked examples

In this subsection, we present two indicative example executions of the main solving algorithm. Example 1 demonstrates the basic mapping for nodes in the follow graph to constraints. The solution requires the simultaneous intersection of several automata. The example is similar in spirit to the core *concat-intersect problem* we introduced in previous work (Hooimeijer and Weimer 2009) associated with the DPRLE tool. As such, the example also serves to highlight the fundamental difference between the older work (an eager algorithm expressed in terms of high-level automata operations) and the algorithm presented in this paper (simultaneous lazy intersection of multiple automata). We discuss this example in detail with reference to line numbers in Fig. 3.

Example 2 illustrates the fact that constraints can be cyclic in nature. In this case, the solution for string variable v_1 depends on the concrete solution for v_2 and vice versa; the follow graph for this constraint system has a cycle. The solution illustrates that it is possible to solve these constraints by selecting a cut of the follow graph. We discuss this example at a slightly higher level, focusing on the automata intersections of interest rather than specific line numbers in the pseudocode of Fig. 3.

Example 1 Consider the example constraint system, as seen before in Sect. 2.2:

$$C_1 = (v_1 \in \mathtt{a}^\star)$$
$$C_2 = (v_2 \in \mathtt{ab})$$
$$C_3 = (v_1 \circ v_2 \in \mathtt{ab})$$

⊢C_1⊣ ⊢C_2⊣
(n_1) → (n_2)
⊢——C_3——⊣

The initial searchstate (generated on line 11 of Fig. 3) would be:

$$\begin{aligned}
&\{\ \ \mathit{next} = \mathsf{nil};\\
&\mathit{states} = \{v_1 \mapsto \{(C_1,1) \mapsto \mathsf{StartsAt}(\mathsf{nfa}(C_1).s);\\
&\qquad\qquad\qquad (C_3,1) \mapsto \mathsf{StartsAt}(\mathsf{nfa}(C_3).s)\};\\
&\qquad\quad v_2 \mapsto \{(C_2,1) \mapsto \mathsf{StartsAt}(\mathsf{nfa}(C_2).s);\\
&\qquad\qquad\qquad (C_3,2) \mapsto \mathsf{Unknown}\ \ \}\}\}
\end{aligned}$$

The main search procedure now visits this searchstate. The visit_state procedure, in turn, calls pick_advance (line 25). We assume $O.\mathit{next}$ is set to v_1, since it has exclusively StartsAt restrictions; we can determine this with a topological sort of the follow graph.

The advance procedure is called to intersect the prefixes of the language for C_1 with the prefixes of the language for C_3. Suppose the intersection (unluckily) results in a path matching a. This replaces the two StartsAt restrictions for v_1 with Path restrictions. On line 26, *restrs* now equals:

$$\begin{aligned}
&\{\ \ (C_1,1) \mapsto \mathsf{Path}([\mathsf{nfa}(C_1).s, \mathsf{nfa}(C_1).s]);\\
&\quad (C_3,1) \mapsto \mathsf{Path}([\mathsf{nfa}(C_3).s, \mathsf{nfa}(C_3).q'])\ \ \}
\end{aligned}$$

$\mathsf{nfa}(C4).q'$ Is some state in $\mathsf{nfa}(C_3)$ reachable on a from $\mathsf{nfa}(C_3).s$.

On lines 29–35, we create the next search state to visit. Because $v_2 \in \mathsf{succ}(v_1, G)$, and v_2 has an Unknown restriction on the correct occurrence, the final O' is:

$$\begin{aligned}
&\{\ \ \mathit{next} = \mathsf{nil};\\
&\mathit{states} = \{v_1 \mapsto \{(C_1,1) \mapsto \mathsf{Path}([\mathsf{nfa}(C_1).s, \mathsf{nfa}(C_1).s]);\\
&\qquad\qquad\qquad (C_3,1) \mapsto \mathsf{Path}([\mathsf{nfa}(C_3).s, \mathsf{nfa}(C_3).q'])\ \ \};\\
&\qquad\quad v_2 \mapsto \{(C_2,1) \mapsto \mathsf{StartsAt}(\mathsf{nfa}(C_2).s);\\
&\qquad\qquad\qquad (C_3,2) \mapsto \mathsf{StartsAt}\ (\mathsf{nfa}(C_3).q')\ \ \}\}\}
\end{aligned}$$

At this point, visit_state returns (line 36) and O' is pushed onto the stack (line 16). On the next iteration, pick_advance selects v_2, since it is the only variable with work remaining. When we call advance, we notice a problem: C_2 requires that v_2 begin with "a", but we have already consumed the "a" in C_3 using v_1. This means no NFA paths are feasible, and we return Back (line 28).

In search, we pop O_{cur} off the stack (line 17). On the next loop iteration, we revisit the initial search state. Since we previously set $O.\mathit{next} \leftarrow v_1$, we proceed immediately to the advance call without calling pick_advance. The advance procedure has only one path left to return: the trivial path that matches the empty string ϵ. At

the end of visit_state, O' now equals:

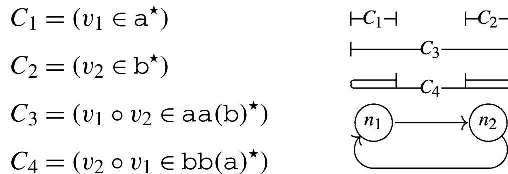
$$\begin{aligned}
\{\ & \mathit{next} = \mathsf{nil}; \\
& \mathit{states} = \{v_1 \mapsto \{(C_1, 1) \mapsto \mathsf{Path}([\mathsf{nfa}(C_1).s]); \\
& \qquad\qquad\qquad (C_3, 1) \mapsto \mathsf{Path}([\mathsf{nfa}(C_3).s])\ \}; \\
& \qquad\quad v_2 \mapsto \{(C_2, 1) \mapsto \mathsf{StartsAt}(\mathsf{nfa}(C_2).s); \\
& \qquad\qquad\qquad (C_3, 2) \mapsto \mathsf{StartsAt}(\mathsf{nfa}(C_3).s)\ \}\}\}
\end{aligned}$$

On the next iteration, pick_advance again selects v_2. A call to advance yields agreeing paths from $\mathsf{nfa}(C_2).s$ to $\mathsf{nfa}(C_2).f$ and from $\mathsf{nfa}(C_3).s$ to $\mathsf{nfa}(C_3).f$. On the final iteration, the all_paths check on line 22 is satisfied, and we extract the satisfying assignment from O_{cur} on line 18.

This example illustrates several key invariants. The algorithm starts exclusively with StartsAt and Unknown restrictions. Each forward step in the search tightens those restrictions by moving from StartsAt to Path and from Unknown to StartsAt. Any given search state is guaranteed to have mutually consistent restrictions. Once set, the only way to eliminate a restriction is by backtracking. Backtracking occurs only if, given the current restrictions, it is impossible to find an agreeing set of paths for the selected variable.

Example 2 In this example we consider a constraint system that imposes cyclic dependencies among two constraints. For brevity, we will elide explicit references to the pseudocode of Fig. 3. Consider the following constraint system, which contains a cyclic order-dependency across two variables:

$$\begin{aligned}
C_1 &= (v_1 \in \mathtt{a}^\star) \\
C_2 &= (v_2 \in \mathtt{b}^\star) \\
C_3 &= (v_1 \circ v_2 \in \mathtt{aa}(\mathtt{b})^\star) \\
C_4 &= (v_2 \circ v_1 \in \mathtt{bb}(\mathtt{a})^\star)
\end{aligned}$$



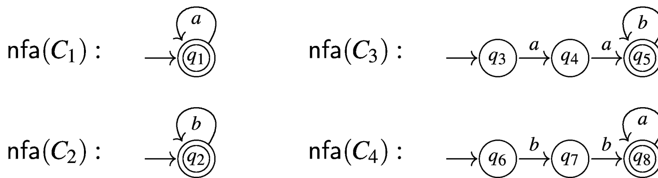
The initial search state for this constraint system is as follows:

$$\begin{aligned}
\{\ & \mathit{next} = \mathsf{nil}; \\
& \mathit{states} = \{v_1 \mapsto \{(C_1, 1) \mapsto \mathsf{StartsAt}(\mathsf{nfa}(C_1).s); \\
& \qquad\qquad\qquad (C_3, 1) \mapsto \mathsf{StartsAt}(\mathsf{nfa}(C_3).s); \\
& \qquad\qquad\qquad (C_4, 2) \mapsto \mathsf{Unknown}\ \}; \\
& \qquad\quad v_2 \mapsto \{(C_2, 1) \mapsto \mathsf{StartsAt}(\mathsf{nfa}(C_2).s); \\
& \qquad\qquad\qquad (C_3, 2) \mapsto \mathsf{Unknown}; \\
& \qquad\qquad\qquad (C_4, 1) \mapsto \mathsf{StartsAt}(\mathsf{nfa}(C_4).s);\ \ \}\}\}
\end{aligned}$$

This state represents a fundamental difference between this example and the previous, non-cyclic, constraint system: both v_1 and v_2 now have an Unknown restriction. This

is because constraints C_3 and C_4 are mutually order-dependent: the algorithm does not know the start state for v_1 because it depends on the path for v_2, and vice versa. This is further apparent from the structure of the follow graph: there is no well-defined topological ordering because nodes n_1 and n_2 form a cycle.

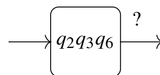
The solution to this problem is conceptually simple: we guess a StartsAt constraint for one of the variables and then conduct the search as previously described. In the example, we could pick any state q in $\mathsf{nfa}(C_4)$ and update the v_1 state to include $(C_4, 2) \mapsto \mathsf{StartsAt}(q)$. If forced to backtrack repeatedly, we will exhaustively consider all other states as potential "guess" candidates; if we rule out all candidates, we conclude that the system is unsatisfiable. For this system, we assume the following NFAs:



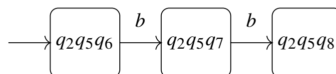
The algorithm randomly selects v_2 to restrict; this corresponds to "cutting" the $n_1 \to n_2$ edge in the follow graph. This means we need to find a start state for occurrence $(C_3, 2)$ of variable v_2. We begin with the start state of $\mathsf{nfa}(C_3)$: state q_3, which yields the following updated search state:

$$\begin{aligned}
&\{\ \mathit{next} = v_2;\\
&\mathit{states} = \{v_1 \mapsto \{(C_1, 1) \mapsto \mathsf{StartsAt}(\mathsf{nfa}(C_1).s);\\
&\qquad (C_3, 1) \mapsto \mathsf{StartsAt}(\mathsf{nfa}(C_3).s);\\
&\qquad (C_4, 2) \mapsto \mathsf{Unknown}\ \};\\
&\quad v_2 \mapsto \{(C_2, 1) \mapsto \mathsf{StartsAt}(\mathsf{nfa}(C_2).s);\\
&\qquad (C_3, 2) \mapsto \mathsf{StartsAt}(q_3 = \mathsf{nfa}(C_3).s);\\
&\qquad (C_4, 1) \mapsto \mathsf{StartsAt}(\mathsf{nfa}(C_4).s);\ \ \}\}\}
\end{aligned}$$

Note that we have updated the restrictions for v_2, and since that variable now has exclusively StartsAt constraints, we are ready to find a path for that variable. Our intersection automaton, denoted by square states, fails immediately, however, because state q_3 has no outbound transitions on b:



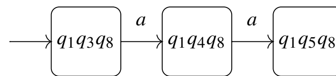
Having failed to find a valid set of Path restrictions for v_2, we select another state in $\mathsf{nfa}(C_3)$, and update the search state accordingly. If we select q_5, our search is more fruitful:



This right-most intersection state is of interest because it represents final states q_5 and q_2 for two constraints (C_2 and C_3) in which v_2 occupies the final position. At this point, we can try to set Path restrictions for v_2 and start the search for path restrictions for v_1:

$$\begin{aligned}
\{\ & \textit{next} = v_1; \\
& \textit{states} = \{v_1 \mapsto \{(C_1, 1) \mapsto \textsf{StartsAt}(\textsf{nfa}(C_1).s); \\
& \qquad\qquad (C_3, 1) \mapsto \textsf{StartsAt}(\textsf{nfa}(C_3).s); \\
& \qquad\qquad (C_4, 2) \mapsto \textsf{StartsAt}(q_8)\}; \\
& \qquad v_2 \mapsto \{(C_2, 1) \mapsto \textsf{Path}([q_2, q_2, q_2]); \\
& \qquad\qquad (C_3, 2) \mapsto \textsf{Path}([q_5, q_5, q_5]); \\
& \qquad\qquad (C_4, 1) \mapsto \textsf{Path}([q_6, q_7, q_8]);\ \}\}\}
\end{aligned}$$

Note that, implicit in the path restrictions for v_2, any solution for v_1 must end in state q_5 for constraint C_3. This is not because state q_5 happens to be a final state; it is specifically necessary because the solution for v_1 must end where the path for v_2 starts. At this point, we do not need to "guess" any states; our only choice is whether to find a longer match for v_2 or start looking for a path for v_1. Since there are no outbound edges on b from q_8, we are forced to choose the latter. The path for v_1 is then as follows:



This yields the following final searchstate:

$$\begin{aligned}
\{\ & \textit{next} = \textsf{nil}; \\
& \textit{states} = \{v_1 \mapsto \{(C_1, 1) \mapsto \textsf{Path}([q_1, q_1, q_1]); \\
& \qquad\qquad (C_3, 1) \mapsto \textsf{Path}([q_3, q_4, q_5]); \\
& \qquad\qquad (C_4, 2) \mapsto \textsf{Path}([q_8, q_8, q_8])\}; \\
& \qquad v_2 \mapsto \{(C_2, 1) \mapsto \textsf{Path}([q_2, q_2, q_2]); \\
& \qquad\qquad (C_3, 2) \mapsto \textsf{Path}([q_5, q_5, q_5]); \\
& \qquad\qquad (C_4, 1) \mapsto \textsf{Path}([q_6, q_7, q_8]);\ \}\}\}
\end{aligned}$$

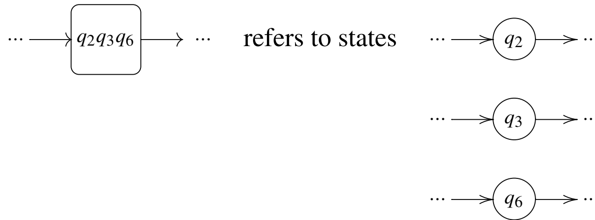
This final state, in turn, yields the satisfying assignment $v_1 = \texttt{aa} \wedge v_2 = \texttt{bb}$.

2.5 Tightening restrictions: the advance procedure

Thus far, we have described the main search algorithm while assuming a procedure for correctly "advancing" variable restrictions from Unknown to StartsAt to Path. We now provide an informal description of the advance function (called on line 26 of Fig. 3), which performs the automaton intersection operations during the search. Given some combination of Unknown, StartsAt, and Path restrictions on the occurrences of a given variable, the goal is to convert every StartsAt restriction to a Path

restriction *while respecting all existing restrictions*. How we conduct the traversal for each variable depends on the restriction types for the variable's occurrences.

In the preceding sections we described automata intersections by multi-labeling the states, for example:



Here, q_2, q_3, and q_6 represent states in distinct (eagerly constructed) input automata for the current constraint system; the square state $q_2q_3q_6$ is an intermediate product state in our lazy search. This section describes the intersection process in more detail.

Recall that we represent a constraint system I as a follow graph G (with one vertex for each variable) and a mapping M between paths in that graph and the constraints they represent (Sect. 2.2). The main search procedure (Sect. 2.3) makes calls to advance, using the *next* field in the current searchstate to signal which variable should be further restricted. Let n_i be a follow graph node that corresponds to variable v_i in a constraint system I. For node $n_i = \mathsf{node}(v_i)$ in the follow graph, we must compute the simultaneous intersection for the set:

$$\mathsf{machines}(n_i) = \left\{ \mathsf{nfa}(R) \mid C \in I = \ldots \diamond R \ \wedge \ M[C] = [\ldots n_i \ldots] \right\}$$

In other words, when $\mathsf{advance}(O, G, M)$ is called with $O.next = v_i$, then our goal is to return a set of new restrictions for v_i by performing a partial simultaneous intersection of the set of automata $N = \mathsf{machines}(\mathsf{node}(v_i))$. We define a square state as a partial mapping Q from NFAs to states; the domain of the partial mapping is N; the mapping has invariant $Q[\mathsf{nfa}(R)] \in \mathsf{states}(\mathsf{nfa}(R))$ for all regular expressions R, where $\mathsf{states}(\cdot)$ denotes the set of states of its parameter. The advance procedure consists of the following steps:

1. Let $O.states[O.next] \ : \ \mathsf{pos} \rightarrow \mathsf{status}$ be the set of occurrences of the current variable, together with the associated restriction for each occurrence. As a precondition, we assume that one of two conditions hold for variable $O.next$: it may have mutually agreeing Path restrictions (in which case no work remains), or it may have some combination of StartsAt and Unknown constraints.
   In the main algorithm, there are two ways for a variable instance to acquire a StartsAt restriction: by being the leftmost variable in a constraint (Fig. 3, line 10), or by immediately following another variable instance that has a Path restriction (lines 32–35). In cyclic cases, however, we are forced to guess at least once. If any restrictions remain Unknown, we guess StartsAt constraints for them and return. The guess can be random, but we must guarantee that repeated calls to advance will enumerate all possible combinations.
2. If $O.states[O.next]$ consists entirely of StartsAt constraints, then we start looking for Path restrictions as follows. The start state for the search is Q_{start} is based

directly on the StartsAt restrictions:

$$Q_{start} = \{\mathsf{nfa}(R) \mapsto q_r \mid \exists o : \mathsf{pos}\ \mathbf{s.t.}\ \mathsf{StartsAt}(\mathsf{nfa}(R).q_r) \in O.states[O.next][o]\}$$

3. We perform simultaneous NFA intersection, incrementally generating a set of Path constraints **P** for all variable occurrences. Each Path constraint is analogous to a projection of the path through the intersection automaton onto a distinct input machine. Our goal is to find a final intersection state that satisfies the following conditions:

   – Let the set of *final* machines for node n_i be:

   $$F = \mathsf{machines}(n_i) \setminus \left(\cap_{n' \in \mathsf{succ}(n_i,G)} \mathsf{machines}(n')\right)$$

   For each automaton X in F, the corresponding Path restriction $P \in \mathbf{P}$ must satisfy $\mathsf{last}(P) = X.f$; i.e., the last state in the path must be final for that specific input machine.

   – Consider the set of StartsAt and Path restrictions for nodes that follow n_i in the follow graph:

   $$\begin{aligned}\{\, x \mid\ & \exists n' \in \mathsf{succ}(n_i),\\ & \exists p\ :\ \mathsf{pos}\ \mathbf{s.t.}\ O.states[\mathsf{var}(n')][p] = x \wedge \\ & x \in \{\mathsf{StartsAt}(\cdot), \mathsf{Path}(\cdot)\}\ \}\end{aligned}$$

   For any machines that are relevant to the current node n_i as well as a successor (i.e., $\mathsf{machines}(n_i) \cap (\cup_{n_i \in \mathsf{succ}(n_i)} \mathsf{machines}(n_i))$), we must ensure that the last state $\mathsf{last}(P)$ for any restriction $\mathsf{Path}(P)$ matches the corresponding starting state for the successor.

4. Finally, if we have exhausted all possible combinations of start and final states for $\mathsf{machines}(n_i)$, we signal failure to the main search algorithm. Note that, holding the restrictions on other nodes constant, the only interactions across follow graph nodes are through shared states.

In summary, advance moves restrictions from Unknown to StartsAt to Path, for one variable at a time. We do this by intersecting the set of automata that affect the possible values for the variable of interest. It is important to note that calls to advance affect each other exclusively through restrictions on NFA states.

2.6 Correctness

Having described our algorithm, we now turn to an informal correctness argument. Decision procedures that return witnesses, in general, are required to be sound, complete, and terminate for all valid inputs. We discuss each of these aspects in turn, referring back to the definitions in Sect. 2.1 and the pseudocode of Fig. 3 when necessary.

Definition 1 (Soundness)

$$\forall I, \mathsf{search}(\mathsf{follow_graph}(I)) = \mathsf{Sat}(A) \Rightarrow \forall (E \diamond R) \in I, [\![E]\!]_A \diamond [\![R]\!]$$

We assume the correctness of the follow_graph procedure. The start_states and visit_state procedures enforce the following invariants for NFA paths:

- The first variable occurrence in each constraint must have its path start with the start state for that constraint's NFA.
- All non-first variable occurrences in each constraint must have their paths start with the final state of their immediate predecessor in the constraint.
- The last variable occurrence in each constraint must have its path end with the final state for that constraint's NFA.

The first bullet is enforced by start_states (as defined in the text) using StartsAt restrictions; these restrictions are preserved when advance moves the StartsAt restrictions to Path restrictions. The second bullet is enforced directly by visit_state in lines 32–35 when moving Unknown restrictions to StartsAt restrictions.The third bullet is enforced by advance when generating paths.

Taken together, these conditions show exactly the right-hand side of the implication: for each constraint $C = (\ldots \diamond R)$, if we concatenate the variable assignments, we end up with a string w that must (by construction) take $\mathsf{nfa}(C).s$ to $\mathsf{nfa}(C).f$, showing $w \diamond R$.

Definition 2 (Completeness)

$$\forall I, \mathsf{satisfiable}(I) \Rightarrow \mathsf{search}(\mathsf{follow_graph}(I)) \neq \mathsf{Unsat}$$

Intuitively, we want to show that for any satisfiable constraint system, there exists a path in a sufficiently-high search tree that reaches an "all paths" searchstate. This argument relies heavily on the completeness of advance, since that procedure essentially determines which child nodes we visit. We provide a high-level overview of that procedure in Sect. 2.5. The completeness is based on three facts:

1. During a single call to advance, we manipulate the search state only with respect to a single variable (or, equivalently, a single node in the follow graph). The only interaction across calls is through the restrictions we apply to start and final states at each follow graph node.
2. Repeated calls to advance, holding all other search state constant, will yield an enumeration of all possible combinations of start and final states in the automata associated with the current variable.
   The machines(...) set of automata that we must intersect is fixed based on the current constraint system, and does not change depending on search state. The maximum number of states in the intersection automaton space is $|NFA|^n$, where $|NFA|$ is an upper bound on the number of states in any input automaton, and $n = |\mathsf{machines}(\ldots)|$.
3. If we exhaust all combinations of start/final states for all variables, then no solution is possible. Even though a single follow graph node may represent infinitely many paths (because of cycles in the intersection automaton), the number of state combinations is necessarily finite, and we can dovetail between finitely many feasible state combinations and their corresponding (potentially infinite) set of paths to enumerate an arbitrary number of satisfying assignments.

Definition 3 (Termination) search returns in a finite number of steps for all inputs.

A termination proof must show that the main loop on lines 13–18 of Fig. 3 always exits in a finite number of steps. This follows from several facts:

- Each vertex in the search tree has a finite number of children, because advance generates paths between a finite number of start states and final states through a cross-product NFA.
- For a given parent vertex in the search tree, we never visit the same child vertex twice. If we backtrack to the parent node, then advance is guaranteed to generate a distinct child node (or report failure).
- The tree has finite height because each step away from the root modifies at least one restriction in the direction of Path. Suppose we assume that all variable occurrences have Unknown restrictions except for one StartsAt restriction (the minimum), and also that we move only one restriction per step. In this case, the maximum height is $\Theta(2n)$ where n is the number of variable occurrences.

3 Experiments

We present several experiments to evaluate the utility of our lazy search approach. In these experiments, we compare the scalability of StrSolve with that of four recently published tools: CFG Analyzer (Axelsson et al. 2008), DPRLE (Hooimeijer and Weimer 2009) and Hampi (Kiezun et al. 2009), and Rex (Veanes et al. 2010; Hooimeijer and Veanes 2011). The experiments are as follows:

- In Sect. 3.1, we consider a benchmark set first used to evaluate Rex (Veanes et al. 2010). Given a pair of regular expressions (a, b), the task is to compute a string in $L(a) \setminus L(b)$, if one exists. The benchmark consists of 10 regular expressions taken from real-world code (Li et al. 2009). We compare DPRLE, Hampi, Rex, and our prototype, running each on all 100 pairs of regular expressions.
- In Sect. 3.2, we take a closer look at the performance characteristics of the Hampi implementation (Kiezun et al. 2009). Internally, Hampi eagerly converts its input constraints to a bitvector formula that is then solved by another solver. This raises an interesting question: how much faster could Hampi be if we swapped out its bitvector solver? In this experiment we re-use the benchmarks from Sect. 3.1 to answer that question.
- In Sect. 3.3, we reproduce and extend an experiment that was first used to evaluate the scalability of the Rex tool (Veanes et al. 2010) relative to the length of the desired string output. For each n between 1 and 1000 inclusive, the task is to compute a string in the intersection of `[a-c]*a[a-c]{n+1}` and `[a-c]*b[a-c]{n}`. We compare DPRLE, Hampi, Rex, and StrSolve.
- In Sect. 3.4, we compare CFG Analyzer, Hampi, and our prototype on a grammar intersection task. We select 85 pairs of context-free grammars from a large data set (Axelsson et al. 2008). The task, for each implementation, is to generate strings of length 5, 10, and 12, in the intersection of each grammar pair.

Across all benchmarks, we use an 8-bit alphabet that corresponds to the extended ASCII character set; we configured all tools to use the same mapping. This is significant because alphabet size can affect performance. The tools were run on the same hardware. The only major difference in configuration was for Rex, which was run under Windows 7 on the same hardware; all other tools were run under a recent Linux configuration.

All experiments were conducted on a 2.8 GHz Intel Core 2 Duo machine with 3.2 GB of addressable RAM. We use unmodified versions of Hampi (revision 24), DPRLE (revision 4), and CFG Analyzer (v. 2007-12-03), all of which are publicly available. We built Hampi from source using Sun Javac (v1.6.0_16); we used the OCaml native compiler (v3.10.2) for CFG Analyzer and DPRLE. We use the prebuilt binaries for STP (Ganesh and Dill 2007) and MiniSAT (Eén and Sörensson 2003) included in the Hampi distribution. We use ZChaff (Moskewicz et al. 2001) (v.2007-03-12) as the underlying SAT solver for CFG Analyzer. StrSolve is written in C++ and built using the GNU C++ compiler (v4.3.3). We measure wall clock time unless otherwise specified. We run Hampi in server mode (Kiezun et al. 2009) to avoid the repeated cost of virtual machine startup unless otherwise specified. Similarly, for Rex we use internal time measurement to avoid measuring virtual machine startup. For CFG Analyzer, DPRLE, and StrSolve, the measured time includes process startup time for each execution.

We used the Rex implementation under license from Microsoft; this tool is not publicly available. The version used was released to us in March 2011. At its core, Rex is an automata library that provides a number of algorithms and data structures; these are described in more detail in Hooimeijer and Veanes (2011). For these experiments, we use a combination of lazy algorithms for intersection and complementation; this is similar in spirit to our own lazy approach, but restricted to single-variable constraints. We note that Hooimeijer and Veanes (2011) focuses exclusively on the time taken by the automata algorithms; we instead measure the tool's full execution time to enable comparison with the other tools.

We use Rex' predicate-based representation for character sets (Pred in Hooimeijer and Veanes 2011). This implementation uses an underlying solver, Z3 (de Moura and Bjørner 2008), to manipulate sets of characters. We assert that the performance of this implementation is indicative for the tool. It is the second-fastest combination reported for a 7-bit alphabet (Hooimeijer and Veanes 2011); our experiments use an 8-bit alphabet for all tools, including Rex. We also conducted runs with Rex' BDD-based implementation, but found that it ran out of memory on a nontrivial number of test cases. We used a recent Microsoft Visual Studio compiler for C# to build and configure the Rex tool in Release mode. When appropriate, we do not measure virtual machine startup for Rex executions; this is analogous to our treatment of Hampi.

3.1 Experiment 1: regular set difference

In this experiment, we test the performance of DPRLE (Hooimeijer and Weimer 2009), Hampi (Kiezun et al. 2009), Rex (Veanes et al. 2010), and StrSolve on a set difference task. We reproduce an experiment originally used to test the symbolic difference construction of Veanes et al. (2010). This experiment uses ten benchmark

<table>
<tr><th></th><th>Regular Expression</th><th>Size</th></tr>
<tr><td>1.</td><td>\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*([,;]\s*\w+([-+.]\w+)*
@\w+([-.]\w+)*\.\w+([-.]\w+)*)*</td><td>1.2 KB</td></tr>
<tr><td>2.</td><td>\$?(\d{1,3},?(\d{3},?)*\d{3}(\.\d{0,2})?|\d{1,3}(\.\d{0,2})?|\.
\d{1,2}?)</td><td>399 B</td></tr>
<tr><td>3.</td><td>([A-Z]{2}|[a-z]{2}[ ]\d{2}[ ][A-Z]{1,2}|[a-z]{1,2}[ ]\d{1,4})?
([A-Z]{3}|[a-z]{3}[ ]\d{1,4})?</td><td>425 B</td></tr>
<tr><td>4.</td><td>[A-Za-z0-9](([\.\-]?[a-zA-Z0-9]+)*)@([A-Za-z0-9]+)
(([\.\-]?[a-zA-Z0-9]+)*)\.[ ]([A-Za-z][A-Za-z]+)</td><td>390 B</td></tr>
<tr><td>5.</td><td>(\w|-)+@((\w|-)+\.)+(\w|-)+</td><td>442 B</td></tr>
<tr><td>6.</td><td>[+-]?([0-9]*\.?[0-9]+|[0-9]+\.?[0-9]*)([eE][+-]?[0-9]+)?</td><td>228 B</td></tr>
<tr><td>7.</td><td>([\w\d.-]+)@{1}(([\w\d-]{1,67})|([\w\d-]+\.[\w\d-]{1,67}))
\.((([a-z]|[A-Z]|\d){2,4})(\.([a-z]|[AZ]|\d){2})?)</td><td>207 KB</td></tr>
<tr><td>8.</td><td>(([A-Za-z0-9]+[_]+)|([A-Za-z0-9]+\-+)|([A-Za-z0-9]+\.+)|
([A-Za-z0-9]+\++))*
[A-Za-z0-9]+@((\w+\-+)|(\w+\.))*\w{1,63}\.[a-zA-Z]{2,6}</td><td>65 KB</td></tr>
<tr><td>9.</td><td>(([a-zA-Z0-9 \-\.]+)@([a-zA-Z0-9 \-\.]+)\.([a-zA-Z]{2,5}){1,25})+
([;.](([a-zA-Z0-9 \-\.]+)@([a-zA-Z0-9 \-\.]+)\.
([a-zA-Z]{2,5}){1,25})+)*</td><td>369 KB</td></tr>
<tr><td>10.</td><td>((\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*)\s*[,]{0,1}\s*)+</td><td>1.3 KB</td></tr>
</table>

Fig. 4 Regular expressions used for Experiment 1. The notation follows that of the .NET framework (Veanes et al. 2010); we use the 8-bit (extended ASCII) interpretation of the character classes (e.g., `\w` and `\d`). The Size column refers to the textual size of the expanded regular expression in the input format for Hampi and STRSOLVE; this requires eliminating repetition operators (curly braces) that are not supported by all tools. Of note is the fact that the sizes vary by several orders of magnitude

regular expressions presented by Li et al. (2009); they are taken from real-world code. The task, for each pair of regular expressions (a, b), is to compute a string that occurs in $L(a)$ but not $L(b)$. This yields 100 distinct inputs for each tool: 90 yes-instances (whenever $a \neq b$) and 10 no-instances (when $a = b$). The regular expressions of interest are listed in Fig. 4.

The majority of the tools under consideration do not natively support repetition operations like `+`, `?`, and `{i,j}`, so we *expand* these operations into the equivalent combination of concatenations and disjunctions (e.g., `a?` becomes `or("", a)` in the input language for Hampi). These expressions are presented in the format used for the Microsoft .NET framework. The *Size* column in Fig. 4 shows the size of each regular expression after expansion. We note that there is a substantial range of sizes: from 228 B (number two) to 369 KB (number nine).

We conducted the experiment as follows. For each pair of expanded regular expressions, we applied the appropriate transformations to create a valid constraint systems for each of the four tools. To facilitate a conservative comparison, this required the following considerations (in each case, giving any potential benefit to the other tool):

– Hampi requires a single fixed length bound for each input, and does not support searching for the empty string. For each pair of input regular expressions, we run Hampi on length bounds 1 through 10, in order, inclusive. We terminate the search

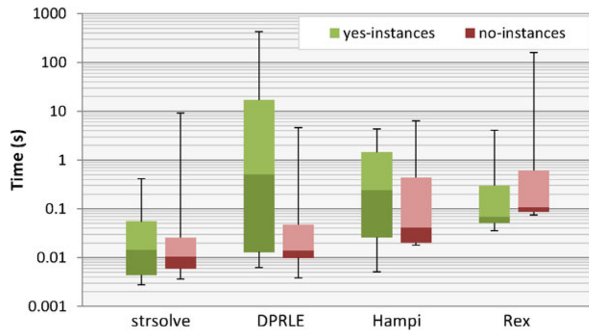


Fig. 5 String generation time distributions (log scale), grouped by yes– and no–instances (left and right of each pair, respectively). The *boxes* represent the 25th through 75th percentile; the *whiskers* represent the 5th through 95th percentile

as soon as Hampi finds a string; this represents a typical usage scenario.[2] In practice, we found that $k = 10$ allowed Hampi to correctly identify all yes-instances.

- DPRLE requires automata descriptions for its input; it does not support regular expressions. Since our prototype performs a conversion from regular expressions to automata, we use that conversion algorithm to generate the DPRLE inputs. We do not count the conversion time towards DPRLE's running time; in practice we found that this made no significant difference.
- Rex uses the .NET regular expression parser and performs its own expansion of repetition operators, so we provide it with the (much smaller) non-expanded regexes. In terms of running time, this represents a trade-off: it saves parsing time at the expense of the time required to perform the expansion (which is not measured for other tools). In practice, we found that running times were dominated by the solving steps and not by the front-end.

Figure 5 summarizes the running times of the tools, grouped by yes-instances (90 datapoints per tool) and no instances (10 datapoints per tool). Note that the median time for our tool on yes-instances is almost an order of magnitude faster than the others, and that our tool exhibits relatively consistent timing behavior compared to all the others (recall log scale when comparing consistency against Rex). The performance gain arises from our construction of the state space corresponding to $L(\overline{b})$: determinization and complementation are performed on this (potentially large) automaton lazily.

3.2 Experiment 2: Hampi's performance

We now take a closer look at the performance breakdown for the Hampi (Kiezun et al. 2009) implementation. Hampi uses a layered approach to solving string constraints; it converts them into bitvector constraints and passes those to an appropriate solver, using the output of that solver to reconstruct the solution. This design allows Hampi to benefit from performance enhancements that may be forthcoming in the area of bitvector constraint solving. In contrast, StrSolve uses specialized algorithms for solving string constraints, and does not stand to benefit from orthogonal research in

[2] Hampi has since added support for ranges of length bounds; at the time of writing, it is implemented using a very similar approach.

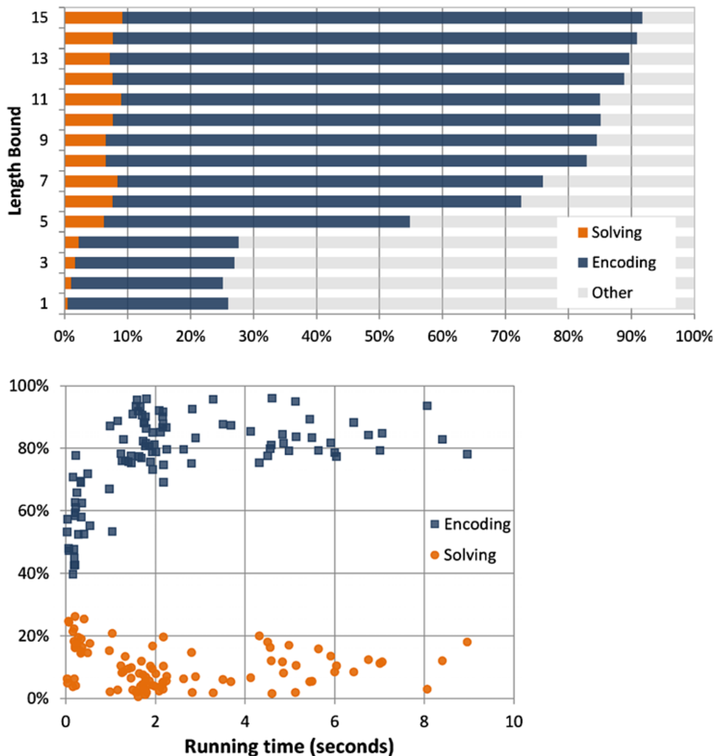


Fig. 6 Hampi execution time breakdown for the dataset of Sect. 3.1. In these graphs, Encoding refers to the process of converting a string constraint system into a bitvector constraint system; Solving refers to the time taken to solve that bitvector constraint system. *On the top graph*, we show the breakdown for length bounds [1; 15]; each *horizontal bar* represents the average of 100 samples. The *bottom graph* shows the relative time spent on encoding and solving for the $k = 15$ case; the *vertical axis* shows percentage of total run time, while the *horizontal axis* represents total solving time

bitvector solving technology. In this experiment, we evaluate whether Hampi could outperform StrSolve given a (hypothetical) faster bitvector solver.

For this experiment, we use the same benchmark set as presented in Sect. 3.1. We instrumented the Hampi source code to add appropriate internal timers for time spent *encoding*, *solving*, and performing all *other* tasks. The timing data is based on 1500 execution runs: 100 runs for 15 distinct length bounds. Figure 6 (left) shows the breakdown for each length bound. The horizontal axis represents the proportion of running time; the vertical axis ranges over length bounds. Figure 6 (right) shows the *encoding* and *solving* measurements for the $k = 15$ length bound, with percentage of total run time on the vertical axis and absolute total running time on the horizontal axis.

These results demonstrate that Hampi's back-end solving step typically accounts for less than 10 % of total execution time. This result illustrates that Hampi's re-encoding step is, by far, the most prominent component of its execution time. In addition, that prominence grows for larger length bounds. Finally, Fig. 6 (right) shows that this is consistently true across test cases, not just when averaging. In fact, the

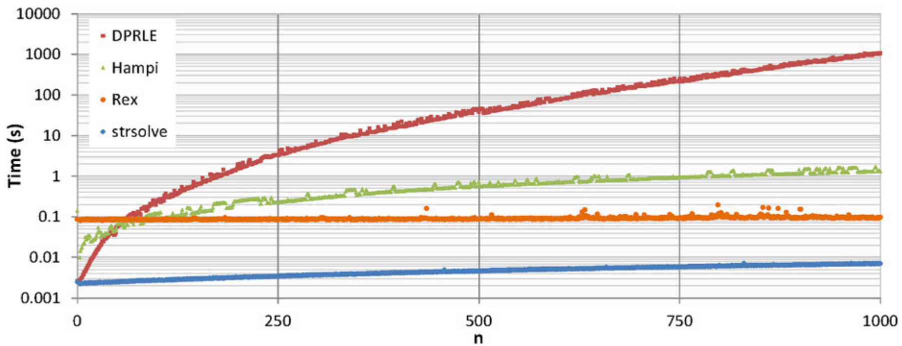


Fig. 7 String generation times (log scale) for the intersection of the regular languages `[a-c]*a[a-c]{n+1}` and `[a-c]*b[a-c]{n}`, for n between 1 and 1000 inclusive

$k = 15$ results suggest that, within this slice of the data, there may exist a positive correlation between total solving time and the proportion of time spent encoding.

At a higher level, these results indicate that Hampi would not be significantly faster if using a faster bitvector solver for these benchmarks. Moreover, for many test cases the encoding time alone exceeds the total time taken by our tool.

3.3 Experiment 3: generating long strings

We hypothesize that our prototype implementation is particularly well-suited for underconstrained systems that require long strings. To test this hypothesis, we reproduce and extend an experiment used to evaluate the scaling behavior of Rex (Veanes et al. 2010). We compare the performance of Hampi, DPRLE, Rex, and STRSOLVE.

The task is as follows. For some length n, given the regular expressions

`[a-c]*a[a-c]{n+1}` and `[a-c]*b[a-c]{n}`

find a string that is in both sets. For example, for $n = 2$, we need a string that matches both `[a-c]*a[a-c][a-c][a-c]` and `[a-c]*b[a-c][a-c]`; one correct answer string is `abcc`. Note that, for any n, the result string must have length $n + 2$. For Hampi, we specify this length bound explicitly; the other tools do not require a length bound.

For each n, we run the four tools, measuring the time it takes each tool to generate a single string that matches both regular expressions. Figure 7 shows our results. Our prototype is, on average, 118× faster than Hampi; the speedup ranges from 4.4× to 239×. DPRLE outperforms Hampi up to $n = 55$, but exhibits considerably poorer scaling behavior than the three other tools. Both STRSOLVE and Rex scale linearly with n, but Rex has a much higher constant cost. Note that, for this experiment, we did not measure virtual machine startup time for Rex.

Finally, an informal review of the results shows that our prototype generates only a fraction of the NFA states; for $n = 1000$, DPRLE generates 1,004,011 states, while our prototype generates just 1,010 (or just 7 more than the length of the discovered path). These results suggest that lazy constraint solving can save large amounts of work relative to eager approaches like Hampi and DPRLE.

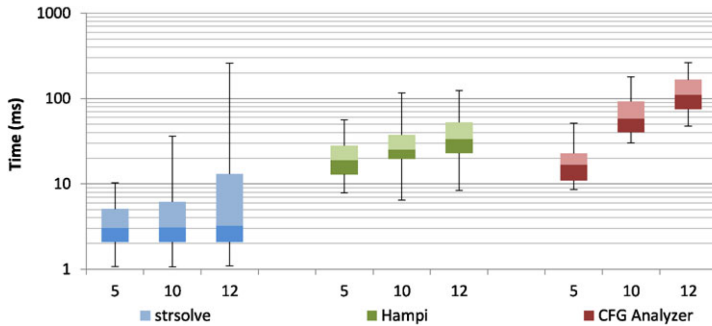


Fig. 8 String generation times (log scale) for the intersection of context-free grammars. The grammar pairs were randomly selected from a dataset by Axelsson et al. (2008). Length bounds are 5, 10, and 12. Each column represents 85 data points; the *bars* show percentile 25 through 75 and the *whiskers* indicate percentile 5 through 95

3.4 Experiment 4: length-bounded context-free intersection

In this experiment, we compare the performance of CFG Analyzer (CFGA) (Axelsson et al. 2008), Hampi (Kiezun et al. 2009), and STRSOLVE. The experiment is similar in spirit to a previously published comparison between Hampi and CFGA: from a dataset of approximately 3000 context-free grammars published with CFGA, we randomly select pairs of grammars and have each tool search for a string in the intersection for several length bounds.

CFGA and Hampi differ substantially in how they solve this problem. Hampi internally generates a (potentially large) regular expression that represents all strings in the given grammar at the given bound. CFGA directly encodes the invariants of the CYK parsing algorithm into conjunctive normal form. For STRSOLVE, we assume a bounding approach similar to that of Hampi. We use an off-the-shelf conversion tool, similar to that used by the Hampi implementation, to generate regular languages. We measure the running time of our tool by adding the conversion time and the solving time.

We randomly selected 200 pairs of grammars. Of these 200 pairs, 88 had at least one grammar at each length bound that produced at least one string. We excluded the other pairs, since they can be trivially ruled out without enumeration by a length bound check. We eliminated an additional three test cases because our conversion tool failed to produce valid output. We ran the three implementations on the remaining 85 grammar pairs at length bounds 5, 10, and 12, yielding 255 datapoints for each of the three tools. The ratio of yes–instances to no–instances was roughly equal. In terms of correctness, we found the outputs of Hampi and our prototype to be in exact agreement.

Figure 8 shows the running time distributions for each tool at each length bound. We note that our performance is, in general, just under an order of magnitude better than the other tools. In all cases, our running time was dominated by the regular enumeration step. We believe a better-integrated implementation of the bounding algorithm would significantly improve the performance for larger length bounds, thus potentially increasing our lead over the other tools.

4 Related work

In this section, we discuss closely related work, focusing on other string decision procedures and client applications.

A number of program analyses have been concerned with the values that string expressions can take on at run-time. Christensen et al. (2003) check the validity of dynamically-generated XML. Similarly, Minamide (2005) uses context-free grammars and finite state transducers to perform basic XHTML validity and cross-site scripting checks. Wassermann and Su (2007) build on Minamide's analysis to detect SQL injection vulnerabilities and cross-site scripting vulnerabilities (Wassermann and Su 2008), by combining it with conservative static taint analysis. The Saner project by Balzarotti et al. (2008) combines a similar static component with an additional dynamic step to find real flaws in sanitizer behavior. At a high level, each of these techniques is an end-to-end analysis with a tightly integrated string model; this paper focuses on providing a string decision procedure that is useful for many client analyses.

Yu et al. (2009a, 2010) similarly use an overapproximating automata-based approach in method similar to abstract interpretation to model both strings and their lengths. We do not present a length abstraction in this paper, and other researchers have pointed out that constraint solving over integers and strings without approximation is undecidable in certain circumstances (Bjørner et al. 2009). Closely related work (Yu et al. 2011) demonstrates the use of string analysis to automatically generate safety checks. Existing string constraint solving work has focused primarily on verification and testcase generation; we believe code synthesis may be an interesting avenue for future string constraint solving work.

Testcase generation is frequently cited as a practical application for string constraint solving. One goal of this line of work is to automatically produce an high-coverage test suite (Lakhotia et al. 2009). Path coverage is achieved by, in essence, computing the path predicates or guards associated with a large number of paths in the program and then treating them as constraints over the input variables. Solving the constraint system yields input variables that cause a given path to be taken. Early tools such as DART (Godefroid et al. 2005) or CUTE (Majumdar and Sen 2007) focused largely on scalar constraints. Lakhotia et al. (2010) provide a survey of symbolic test generation. It has since become clear that string analyses are at the heart of the problem: "Test case generation for web applications and security problems requires solving string constraints and combinations of numeric and string constraints" (Cadar et al. 2011, Sect. 4). More recent work has thus focused on the integration of string reasoning into such frameworks (e.g., Godefroid et al. 2008a; Majumdar and Xu 2007). A strong example is that of the Symbolic PathFinder (Pasareanu et al. 2008), which has been extended with a symbolic string analysis by Fujitsu and applied to the testing of web applications (Fujitsu Laboratories 2010).

The Hampi tool (Kiezun et al. 2009) is a solver for string constraints over fixed-size string variables. It supports regular languages, fixed-size context-free languages, and a number of operations (e.g., union, concatenation, Kleene star). The Kaluza project (Saxena et al. 2010) uses Hampi internally, and adds support for multivariate constraints and length constraints. Our decision procedure does not support similar

length constraints. Where features overlap, however, we show our decision procedure to be several orders of magnitude faster than Hampi on indicative workloads.

There are several other approaches that perform a re-encoding from string constraints into another logic. The CFG Analyzer tool (Axelsson et al. 2008) is a solver for bounded versions of otherwise-undecidable context-free language problems. Problems such as inclusion, intersection, universality, equivalence and ambiguity are handled via a reduction to satisfiability for propositional logic in the bounded case. The Rex tool (Veanes et al. 2010) solves string constraints through a symbolic encoding of finite state automata into Z3 SMT solver (de Moura and Bjørner 2008). An important benefit of this strategy is that string constraints can be readily integrated with other theories (e.g., linear arithmetic) handled by Z3. Similarly, Tateishi et al. (2011) propose an encoding to monadic second-order logic (M2L), and implement their technique using the MONA solver for that logic (Henriksen et al. 1995).

The DPRLE tool (Hooimeijer and Weimer 2009) is a decision procedure for regular language constraints involving concatenation and subset operations. The tool focuses on generating entire sets of satisfying assignments rather than single strings: often constraints over multiple variables can yield multiple disjoint solution sets. The core algorithm of DPRLE has been formally proved correct in a constructive logic framework. Our new procedure supports similar operations to those allowed by DPRLE, but efficiently produces single witnesses rather than atomically generating entire solution sets. Nevertheless, our worst-case performance corresponds to that of DPRLE. For a large class of no–instances in which the contradiction occurs close to a right-most variable, our current algorithm necessarily generates a large subset of the NFA states that DPRLE generates by default.

In recent work, we evaluated various datastructures to represent character sets in string constraint solving (Hooimeijer and Veanes 2011). This work finds that, for large alphabets, a representation based on *binary decision diagrams* (BDDs; Bryant 1986) is most performant, while smaller alphabets (such as extended ASCII) see relatively little difference between a range-based implementation (similar to the one used in our tool) and BDDs. This is an interesting finding, and one that we may incorporate into StrSolve in the future. It should be noted that the BDD representation discussed by Hooimeijer and Veanes (2011) is distinct from the BDD representation used by the Mona tool (Møller and Schwartzbach 2001, Sect. 5).

The recent Bek project examines the use of symbolic finite state transducers (Hooimeijer et al. 2011; Veanes et al. 2012) as a model for string-manipulating code. Unlike traditional string analysis work, which aims to model general-purpose code by approximation, Bek instead models a restricted domain-specific language without approximation. The analysis supports deep semantic checks on programs, including program equivalence. The Bek project can be characterized as a constraint solver in which the variables represent code (i.e., input-output relations on strings). Fu et al. provide a transducer model for Perl-style regex replacement operations (Fu and Li 2010; Fu et al. 2012). This type of operation is difficult to model because the semantics are subtle across, for example, eager vs. non-eager replacement. It would be interesting to combine transducer-based analyses with a string constraint solver.

5 Conclusion

Recent work on the analysis of string values has focused on providing external decision procedures for theories that model common programming idioms involving strings. Thus far, this work has focused on features such as support for concatenation operations (Hooimeijer and Weimer 2009), embedding into SMT solvers (Veanes et al. 2010), and bounded context-free languages (Kiezun et al. 2009).

In this paper, we present a constraint-solving algorithm for equations over string variables. Our algorithm has similar features to existing string decision procedures, but is designed to yield faster answers to yes-instances for large input constraint systems. We achieve this by treating the constraint solving problem as an explicit search problem. A key feature of our algorithm is that we instantiate the search space in an on-demand fashion.

We evaluated our algorithm by comparing our prototype implementation to publicly available tools like CFGA (Axelsson et al. 2008), DPRLE (Hooimeijer and Weimer 2009), Rex (Veanes et al. 2010) and Hampi (Kiezun et al. 2009). We used several sets of previously published benchmarks (Kiezun et al. 2009; Veanes et al. 2010); the results show that our approach is up to four orders of magnitude faster than the other tools. We believe that as string constraint solvers continue to become more and more useful to other program transformations and analyses, scalability will be of paramount importance, and our algorithm is a step in that direction.

References

Axelsson, R., Heljanko, K., Lange, M.: Analyzing context-free grammars using an incremental sat solver. In: International Colloquium on Automata, Languages and Programming, pp. 410–422 (2008). doi:10.1007/978-3-540-70583-3_34

Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Saner: composing static and dynamic analysis to validate sanitization in web applications. In: IEEE Symposium on Security and Privacy, pp. 387–401 (2008)

Bjørner, N., Tillmann, N., Voronkov, A.: Path feasibility analysis for string-manipulating programs. In: Tools and Algorithms for the Construction and Analysis of Systems (2009)

Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. **35**(8), 677–691 (1986)

Cadar, C., Godefroid, P., Khurshid, S., Pasareanu, C.S., Sen, K., Tillmann, N., Visser, W.: Symbolic execution for software testing in practice: preliminary assessment. In: International Conference on Software Engineering, pp. 1066–1071 (2011)

Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: International Symposium on Static Analysis, pp. 1–18 (2003)

de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems (2008)

Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3), 365–473 (2005). doi:10.1145/1066100.1066102

Eén, N., Sörensson, N.: An extensible sat-solver. In: Theory and Applications of Satisfiability Testing, pp. 502–518 (2003)

Fu, X., Li, C.C.: Modeling regular replacement for string constraint solving. In: Muñoz, C. (ed.) Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215, NASA, Langley Research Center, Hampton, VA 23681-2199, USA, pp. 67–76 (2010)

Fu, X., Powell, M., Bantegui, M., Li, C.C.: Simple linear string constraints. Form. Asp. Comput. 1–45 (2012). doi:10.1007/s00165-011-0214-3

Fujitsu Laboratories: Fujitsu develops technology to enhance comprehensive testing of Java programs (2010). URL http://www.fujitsu.com/global/news/pr/archives/month/2010/20100112-02.html

Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Computer-Aided Verification, pp. 519–531 (2007)

Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Programming Language Design and Implementation (2005)

Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. In: Programming Language Design and Implementation (2008a)

Godefroid, P., Levin, M., Molnar, D.: Automated whitebox fuzz testing. In: Network Distributed Security Symposium (2008b)

Henriksen, J., Jensen, J., Jørgensen, M., Klarlund, N., Paige, B., Rauhe, T., Sandholm, A.: Mona: monadic second-order logic in practice. In: TACAS ’95. LNCS, vol. 1019. Springer, Berlin (1995)

Hooimeijer, P., Veanes, M.: An evaluation of automata algorithms for string analysis. In: Verification, Model Checking, and Abstract Interpretation, pp. 248–262 (2011)

Hooimeijer, P., Weimer, W.: A decision procedure for subset constraints over regular languages. In: Programming Languages Design and Implementation, pp. 188–198 (2009)

Hooimeijer, P., Weimer, W.: Solving string constraints lazily. In: Automated Software Engineering, pp. 377–386 (2010)

Hooimeijer, P., Livshits, B., Molnar, D., Saxena, P., Veanes, M.: Fast and precise sanitizer analysis with bek. In: USENIX Security Symposium, pp. 1–15 (2011)

Ilie, L., Yu, S.: Follow automata. Inf. Comput. **186**(1), 140–162 (2003). doi:10.1016/S0890-5401(03)00090-7

Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: Hampi: a solver for string constraints. In: International Symposium on Software Testing and Analysis, pp. 105–116 (2009)

Lakhotia, K., McMinn, P., Harman, M.: Handling dynamic data structures in search based testing. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1759–1766 (2008)

Lakhotia, K., McMinn, P., Harman, M.: Automated test data generation for coverage: haven’t we solved this problem yet? In: Testing Academia and Industry Conference, pp. 95–104 (2009)

Lakhotia, K., McMinn, P., Harman, M.: An empirical investigation into branch coverage for c programs using cute and Austin. J. Syst. Softw. **83**(12), 2379–2391 (2010)

Li, N., Xie, T., Tillmann, N., de Halleux, J., Schulte, W.: Reggae: automated test generation for programs using complex regular expressions. In: Automated Software Engineering Short Paper (2009)

Majumdar, R., Sen, K.: Hybrid concolic testing. In: International Conference on Software Engineering, pp. 416–426 (2007)

Majumdar, R., Xu, R.G.: Directed test generation using symbolic grammars. In: Automated Software Engineering, pp. 134–143 (2007)

Minamide, Y.: Static approximation of dynamically generated web pages. In: International Conference on the World Wide Web, pp. 432–441 (2005). http://doi.acm.org/10.1145/1060745.1060809

Møller, A., Schwartzbach, M.I.: The pointer assertion logic engine. In: Programming Language Design and Implementation, pp. 221–231 (2001)

Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient sat solver. In: Design Automation Conference, pp. 530–535 (2001)

Necula, G.C.: Proof-carrying code. In: Principles of Programming Languages, pp. 106–119 (1997)

Pasareanu, C.S., Mehlitz, P.C., Bushnell, D.H., Gundy-Burlet, K., Lowry, M.R., Person, S., Pape, M.: Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In: International Symposium on Software Testing and Analysis, pp. 15–26 (2008)

Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for javascript. In: IEEE Symposium on Security and Privacy, pp. 513–528 (2010)

Sipser, M.: Introduction to the Theory of Computation, 2nd edn. Course Technology, Independence (1997)

Su, Z., Wassermann, G.: The essence of command injection attacks in web applications. In: Principles of Programming Languages, pp. 372–382 (2006)

Tateishi, T., Pistoia, M., Tripp, O.: Path- and index-sensitive string analysis based on monadic second-order logic. In: ISSTA ’11, pp. 166–176. ACM, New York (2011)

Veanes, M., de Halleux, P., Tillmann, N.: Rex: symbolic regular expression explorer. In: International Conference on Software Testing, Verification and Validation, pp. 498–507 (2010)

Veanes, M., Hooimeijer, P., Livshits, B., Molnar, D., Bjørner, N.: Symbolic finite state transducers: algorithms and applications. In: Principles of Programming Languages, pp. 137–150 (2012)

Wassermann, G., Su, Z.: Sound and precise analysis of web applications for injection vulnerabilities. In: Programming Languages Design and Implementation, pp. 32–41 (2007)

Wassermann, G., Su, Z.: Static detection of cross-site scripting vulnerabilities. In: International Conference on Software Engineering (2008)
Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: International Conference on Software Engineering, pp. 364–374 (2009)
Xie, Y., Aiken, A.: Saturn: a SAT-based tool for bug detection. In: Computer Aided Verification, pp. 139–143 (2005)
Xie, Y., Aiken, A.: Static detection of security vulnerabilities in scripting languages. In: USENIX Security Symposium, pp. 179–192 (2006)
Yu, F., Alkhalaf, M., Bultan, T.: Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses. In: Automated Software Engineering, pp. 605–609 (2009a)
Yu, F., Bultan, T., Ibarra, O.H.: Symbolic string verification: combining string analysis and size analysis. In: Tools and Algorithms for the Construction and Analysis of Systems (2009b)
Yu, F., Bultan, T., Ibarra, O.H.: Relational string verification using multi-track automata. In: Conference on Implementation and Application of Automata, pp. 290–299 (2010)
Yu, F., Alkhalaf, M., Bultan, T.: Patching vulnerabilities with sanitization synthesis. In: International Conference on Software Engineering, pp. 251–260 (2011)