

Decision Procedures for String Constraints

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Pieter Hooimeijer

May 2012

Abstract

String-related defects are among the most prevalent and costly in modern software development. For example, in terms of frequency, cross-site scripting vulnerabilities have long surpassed traditional exploits like buffer overruns. The state of this problem is particularly disconcerting because it does not just affect legacy code: developing web applications today — even when adhering to best practices and using modern library support — remains error-prone.

A number of program analysis approaches aim to prevent or mitigate string-related defects; examples include static bug detectors and automated testcase generators. Traditionally, this work has relied on built-in algorithms to reason about string-manipulating code. This arrangement is suboptimal for two reasons: first, it forces researchers to re-invent the wheel for each new analysis; and second, it does not encourage the independent improvement of domain-specific algorithms for handling strings.

In this dissertation, we present research on specialized decision algorithms for string constraints. Our high-level approach is to provide a constraint solving interface; a client analysis can use that interface to reason about strings in the same way it might use a SAT solver to reason about binary state. To this end, we identify a set of string constraints that captures common programming language constructs, and permits efficient solving algorithms. We provide a core solving algorithm together with a machine-checkable proof of its correctness.

Next, we focus on performance. We evaluate a variety of datastructures and algorithms in a controlled setting to inform our choice of each. Our final approach is based on two insights: (1) string constraints can be cast as an explicit search problem, and (2) to solve these constraints, we can

instantiate the search space lazily through incremental refinement. These insights lead to substantial performance gains relative to competing approaches; our experimental results show our prototype to be several of magnitude faster across several published benchmarks.

Approval Sheet

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)



Pieter Hooimeijer

This dissertation has been read and approved by the Examining Committee:



Westley Weimer, Adviser



David Evans, Committee Chair



Michael Hill

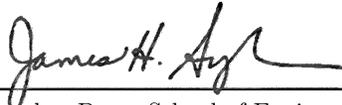


Sriram Rajamani



Alfred Weaver

Accepted for the School of Engineering and Applied Science:



James Aylor, Dean, School of Engineering and Applied Science

May 2012

Chapter 1

Introduction

This dissertation focuses on a common source of software defects: string manipulation. String-related defects are among the most prevalent and costly in modern software development. They are typically caused by the improper handling of structured text such as HTML, XML, and SQL [1, 2]. Two compelling examples of this type of defect are SQL injection and cross-site scripting vulnerabilities. These vulnerabilities are common; together they accounted for 35.5% of reported security vulnerabilities in 2006 [3]. A November 2009 study found that 64% of the 1,364 surveyed websites had at least one serious vulnerability [4], where *serious* means “Exploitation could lead to serious and direct business impact.”

Reasoning about strings is a key aspect in many types of program analysis work, including static bug finding [1, 5, 6, 7] and automated testing [8, 9, 10, 11, 12]. Until recently, this work has relied on ad hoc algorithms to formally reason about the values that string variables may take at runtime. That situation is suboptimal for two reasons: first, it forces researchers to re-invent the wheel for each new tool; and second, it does not encourage the independent improvement of domain-specific reasoning for strings.

In this dissertation, we focus on the development of algorithms that enable formal reasoning about string operations; we refer to these algorithms as *string decision procedures*. Informally, a *decision procedure* is an algorithm that, given an input formula, answers the yes-or-no question:

“Does this formula express a consistent set of constraints?” The use of decision procedures in program analysis is becoming pervasive: the practice of querying an external algorithm to reason about pointer aliasing [13, 14, 15] is a standard example. When using a decision procedure, the set of input formulae and their semantics are referred to as a *theory*; commonly used theories include uninterpreted functions and equality, boolean satisfiability (SAT), linear arithmetic, and bitvector logic. *Satisfiability modulo theories* (SMT) solvers provide a generalization of SAT that includes theories and first-order logic [16, 17]. Unfortunately, existing theories do not facilitate direct reasoning about string-manipulating code.

Our goal is to develop tools that can be used by a wide variety of client analyses to reason about code that includes common high-level string operations, in the same way they might use a boolean satisfiability (SAT) solver to reason about boolean state. This leads to the following thesis statement:

It is possible to construct a practical algorithm that decides the satisfiability of constraints that cover both string and integer index operations, scales up to real-world program analysis problems, and admits a machine-checkable proof of correctness.

Based on this thesis, we address the following challenges:

- **Expressive Utility.** String decision procedures cannot efficiently support all common string operations.¹ In this dissertation, we select a core set of operations and demonstrate empirically that the corresponding constraint language is sufficiently expressive to model real-world programming analysis problems, like extending a static bug finder to generate testcases (Chapter 2).
- **Scalability.** Good performance is crucial if string decision procedures are to see real use by client analyses. One of the direct benefits of creating specialized algorithms lies in the fact that we can then apply domain-specific optimizations. In this dissertation, we provide an apples-to-apples performance comparison of candidate automata datastructures and algorithms Chapter 3. This comparison informs an optimized version of our core algorithm, which we present in Chapter 4.

We implement this algorithm in a tool called STRSOLVE, and evaluate its performance relative

¹In the extreme, consider a hash function like SHA-2, which is based on a block cipher. If included in a constraint language, the corresponding solving algorithm would have to efficiently consider all possible inputs (of many possible lengths) for a given (fixed-length) output; this is believed to be infeasible in practice.

to a number of other approaches, and find that and demonstrate that it is several orders of magnitude faster than other tools on indicative benchmarks.

- **Correctness.** Decision procedures are particularly well-suited for formal correctness arguments because their correctness conditions (soundness and completeness) permit succinct formal descriptions. We believe correctness arguments are helpful in this context, in particular when developing formal problem descriptions and solving algorithms in tandem. In this dissertation, we render a machine-verifiable proof of correctness for the core algorithm described in Chapter 2.

Over the course of following chapters, we focus on different subsets of these challenges. In the Chapter 2, we emphasize correctness and expressiveness. Our correctness proof, rendered in the calculus of inductive constructions [18], describes characters, strings, automata, and automata algorithms from first principles. Chapter 3 addresses scalability, in particular the efficiency of low-level automata operations. Chapter 4 builds on the results of Chapters 2 and 3, by providing a more general solving algorithm and focusing in roughly equal parts on expressiveness (comparing against a number of other approaches), scalability (applying results from Chapter 3 in a broader setting), and correctness (a proof sketch). The main contributions of this dissertation are as follows:

1. The identification and formal definition of the *Regular Matching Assignments* (RMA) problem (Chapter 2).
2. An automata-based algorithm, `concat_intersect`, its correctness proof rendered in the calculus of inductive constructions [18], and an implementation (Decision Procedure for Regular Language Equations (DPRLE); Chapter 2).
3. The evaluation of the expressive utility of (2) in the context of generating testcases that trigger SQL injection vulnerabilities in a corpus of real-world PHP code (Chapter 2).
4. An apples-to-apples performance comparison of datastructures and algorithms for automata-based string constraint solving (Chapter 3). We pay special attention to isolating the core operations of interest, and use a popular open source implementation as a performance baseline.

5. A novel decision procedure that supports the efficient and lazy analysis of string constraints (Chapter 4). 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.
6. A comprehensive performance comparison between our STRSOLVE prototype (5) and implementations from related work (Chapter 4). 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.

This dissertation is structured as follows. The remainder of this chapter discusses the background of decision procedures in program analysis in Section 1.1 and provides a motivating example to illustrate the potential benefits of providing similar decision procedures for strings in Section 1.2. Chapters 2–4 present our main contributions. Chapter 5 discusses closely related work. We provide conclusions and directions for future work in Chapter 6.

1.1 Decision Procedures in Program Analysis

Having outlined the dissertation, we now provide a brief overview of the use of external decision procedures in program analysis. At a high level, program analysis research aims to reduce the costs of software engineering and maintenance by allowing developers to gain confidence about the correct behavior of their code. Work in this area uses a wide variety of approaches to accomplish this; for example, a static technique (e.g., symbolic software model checking [19]) might provide a correctness proof relative to a partial correctness specification, while a dynamic technique (e.g., directed automated random testing [9]) might improve code coverage or eliminate redundant testcases. Many other methods of classification apply. Despite their variety, the majority of end-to-end program analysis tools share a set of core algorithms. Many of these core algorithms were originally developed as part of a particular end-to-end analysis, and later separated out and further developed when their wider utility was recognized. The canonical example is points-to analysis (e.g., [13]); most

mainstream programming languages feature heap-allocated data and pointers, and many analysis tools use an off-the-shelf algorithm to find (or rule out) potential aliasing between pointers.

Many core analysis algorithms can be cast as a decision procedure; in practice, that term is interchangeable with *constraint solver*. In this context, performing alias analysis is analogous to solving a system of set constraints [20]. More direct examples include solvers for boolean satisfiability (SAT solvers) [21, 22, 23] and, more recently, satisfiability modulo theories (SMT solvers) [16, 24, 25, 26], which support the integration of decision procedures for multiple theories. SMT solvers are typically used to test whether a given set of assumptions is internally (logically) consistent. In general, they take as input a constraint system over some number of free variables, and return either **Satisfiable** if the constraints can be solved, or **Unsatisfiable** if no solution exists. For **Satisfiable** instances, we can typically extract concrete values (i.e., witnesses) for the free variables.

Modern SMT solvers support a number of different constraint types, referred to as *theories*, and many of those theories are geared specifically to program analysis. As an informal example, consider the following two mathematical constraints:

$$x^2 = 25$$

$$x > 0$$

where x is an integer. We might generate such a constraint system while analyzing code that branches on a similar set of conditions, e.g., `if (x > 0 && x*x == 25) { ... }`.

Solving this constraint system involves (1) deciding whether a solution exists, and (2) if yes, finding a concrete value for x . In this case, the constraint system is satisfiable: the two constraints on x are consistent if and only if $x = 5$. To arrive at that solution automatically, we can express this constraint system directly to an SMT solver as follows.²

Example 1.1. *Nonlinear arithmetic using an SMT solver:*

²Using the SMT-LIB 2.0 standard format.

```
1 (declare-fun x () Int)
2 (assert (= (* x x) 25))
3 (assert (> x 0))
4 (check-sat)
5 (get-model)
```

We declare x as a nullary function (i.e., a variable) with domain `Int` (line 1), then `assert` the individual constraints (lines 2–3), and finally ask for both a satisfiability check (line 4) and a concrete value for x (line 5). The resulting output from an SMT solver, in this case Microsoft’s Z3 [16], is:

```
1 sat
2 (model
3 (define-fun x () Int
4 5)
5 )
```

This gives us the expected results: the constraints system is satisfiable (line 1), and 5 is a satisfying assignment for x . ☒

Depending on the context, we may prefer to model integers as bitvectors rather than mathematical integers, since the `Int` type does not model overflow. The following example illustrates using an SMT solver that, when using bitvectors, it is possible to overflow and arrive at $w_1 \cdot w_2 = 0$ even if neither w_1 nor w_2 are zero.

Example 1.2. *Bitvectors and overflow using an SMT solver:*

```

1 (declare-fun w1 () (_ BitVec 8))
2 (declare-fun w2 () (_ BitVec 8))
3 (declare-fun w3 () (_ BitVec 8))
4
5 (assert (= w1 #x02))
6 (assert (= w3 #x00))
7 (assert (= w3 (bvmul w1 w2)))
8 (assert (not (= w2 #x00)))
9
10 (check-sat)
11 (get-model)

```

Here, we declare three 8 bit wide variables `w1–w3` (lines 1–3); we will interpret them as unsigned integers in twos-complement notation. We set `w1` to a fixed value 2 (line 5, in hexadecimal), and constrain `w3` to 0 (line 6) and the twos-complement multiplication of `w1` by `w2` (line 7). Finally, we assert that `w2` cannot be zero (line 8). The corresponding output is:

```

1 sat
2 (model
3   (define-fun w2 () (_ BitVec 8)
4     #x80)
5   (define-fun w3 () (_ BitVec 8)
6     #x00)
7   (define-fun w1 () (_ BitVec 8)
8     #x02)
9 )

```

Note that `w2` gets concrete value 128 (in decimal), yielding 256 for `w3` — the first value not representable in an 8-bit wide variable. ☒

In the context of program analysis, constraint solving algorithms for decidable theories are commonly referred to as *decision procedures*; or *semi-decision procedures* for undecidable theories.

Example 2 shows the output of Z3's decision procedure for quantifier-free theory of bitvectors; the theory is decidable and NP-Complete by reduction from 3-SAT. Example 1 demonstrates the use of a semi-decision procedure for nonlinear integer arithmetic, a theory for which the satisfaction problem is undecidable in general. This means that the underlying (semi-)algorithm can return **Unknown** in addition to **Satisfiable** and **Unsatisfiable**, and may fail to terminate on some inputs. Modern constraint solving tools are the workhorse of many current program analysis projects. It is not uncommon for client analyses to generate problems that include tens of thousands of constraints and variables (e.g., [27, 28]).

1.2 Motivating Example

To motivate the need for string-specific analysis tools, we now turn to an example of an actual SQL injection vulnerability. Figure 1.1 shows a code fragment adapted from Utopia News Pro, a news management web service written in PHP. The `$_POST` array holds values that are submitted by the user as part of an HTTP request. A number of static analyses will (correctly) detect a potential vulnerability on line 7. The check on line 2 is designed to limit `$newsid` to numbers: `[\d]+` is a regular expression for a non-empty sequence of consecutive digits.

The `preg_match` function recognizes the delimiters `$` and `^` to match the end and the beginning of the string respectively. However, the check on line 2 is missing the `^` marker. Thus it is possible that the query sent on line 7 might be, for example, `"SELECT * from 'news' WHERE newsid='nid_' OR 1=1 ; DROP 'news' -- 9"`. That particular query returns all entries in the `news` table to the attacker, and then deletes the table (the `--` begins a comment in SQL). Although the vulnerability is real, it may not be obvious to developers how an untrusted user can trigger it. For example, setting `posted_newsid` to `"' OR 1=1 ; DROP 'news' --"` fails to trigger it, instead causing the program to exit on line 4.

Conventional development relies heavily on regression testing and reproducible defect reports; a testcase demonstrating the vulnerability makes it more likely that the defect will be fixed [29, 30]. We

```

1  $newsid = $_POST['posted_newsid'];
2  if (!preg_match('/[\d]+$/', $newsid)) {
3      unp_msgBox('Invalid article news ID.');
```

Figure 1.1: SQL code injection vulnerability example adapted from Utopia News Pro. The `$_POST` mapping holds untrusted user-submitted data.

therefore wish to form a testcase that exhibits the problem by generating values for input variables, such as:

```

posted_newsid = ' OR 1=1 ; DROP 'news' -- 9
posted_userid = a
```

To find this set of input values, we consider the constraints imposed by the code:

- The input set must pass the (incomplete) safety check on line 2.
- The input set must result in an exploit of interest on line 7.

This problem can be phrased as a constraint system over the string variables. The particular actions taken by the generated exploit (e.g., whether all entries are returned or a table is dropped or modified) are a secondary concern. Instead, we want to allow analyses to detect the problem and generate a test case that includes string input values and a viable execution path through the program that triggers the vulnerability. Finally, note that if the program in Figure 1.1 were fixed to use proper filtering, our algorithm would indicate that language of vulnerable strings for `posted_userid` is empty (i.e., that there is no vulnerability).

Chapter 2

Constraints over Regular Languages

In this chapter, we present a novel decision procedure for solving equations over sets of strings. Our decision procedure computes satisfying assignments for systems of equations that include concatenation, language inclusion constraints, and variables that represent regular sets. The algorithm can be used as a constraint solver to “push back” constraints across string operations. Common string operations, in particular equality checks, length checks against a constant, and regular expression matches, can be translated directly into a special type of constraint that we introduce and define. In this chapter, we focus initially on a formal exposition of the problem, the decision procedure, its soundness and completeness, and its runtime complexity (Sections 2.1–2.3). To evaluate our approach, we implement our algorithm as a standalone tool and use an off-the-shelf symbolic execution framework [7] to generate string constraints for potential SQL injection vulnerabilities (Section 2.4).

The main contributions of this chapter are:

1. The identification and formal definition of the *Regular Matching Assignments* (RMA) problem (Section 2.1).

S	$::= E \subseteq C$	subset constraint
E	$::= E \circ T$	language concatenation
	$ T$	
T	$::= C$	atoms
	$ V$	
C	$::= c_1 \dots c_n$	constants
V	$::= v_1 \dots v_m$	variables

Figure 2.1: Grammar for subset constraints over regular languages. The right-hand side (some element of C) is a single constant. The constants $c_1 \dots c_n$ and variables $v_1 \dots v_m$ each represent a regular language. The goal is to find satisfying assignments for the variables.

2. An automata-based algorithm, `concat_intersect` (Section 2.2), its correctness proof rendered in the calculus of inductive constructions [18], and an implementation (Decision Procedure for Regular Language Equations (DPRLE)).
3. The evaluation of the expressive utility of (2) in the context of generating testcases that trigger SQL injection vulnerabilities in a corpus of real-world PHP code (Section 2.4).

2.1 The Regular Matching Assignments Problem

In the following sections, we present our decision procedure for subset and concatenation constraints over regular languages. In Section 2.1, we provide a formal problem definition. The *Regular Matching Assignments (RMA)* problem defines the language constraints of interest, and also what constitutes a satisfying assignment for such a system of equations. Next, in Section 2.2, we define the *Concatenation-Intersection (CI)* problem, which we show to be a subclass of RMA. We provide an algorithm for solving instances of CI, and prove it correct in Section 2.2.1. In Section 2.3, finally, we extend the CI algorithm to general instances of RMA.

In this section we define the *Regular Matching Assignments (RMA)* problem. The goal is to find satisfying assignments for certain systems of equations over regular languages. These systems consist of some number of constant languages and some number of language variables, combined using subset constraints and language concatenation. Figure 2.1 provides the general form of the constraints of interest.

Example 2.1. *String constraints from code.* The example introduced in Section 1.2 can be expressed as the following set of constraints:

$$v_1 \subseteq c_1 \quad c_2 \circ v_1 \subseteq c_3$$

where c_1 corresponds to the input filtering on line 2, c_2 corresponds to the string constant `nid_` on line 6, and c_3 corresponds to undesired SQL queries. If we solve this system, then variable v_1 is the set of user inputs that demonstrate the vulnerability. If this set is empty, then the code is not vulnerable. If v_1 is nonempty, then we can use it to understand the problem and to generate testcases. ⊠

Definition 2.1. *Assignment.* We write $A = [v_1 \leftarrow x_1, \dots, v_m \leftarrow x_m]$ for an assignment of regular languages $\{x_1, \dots, x_m\}$ to variables $\{v_1, \dots, v_m\}$; let $A[v_i] = x_i$. ⊠

The semantics of expressions are as follows. Let $\llbracket c_i \rrbracket$ be the regular set denoted by language constant c_i (derivable from C in Figure 2.1). For a given assignment A , where A includes a mapping for v_i , let $\llbracket v_i \rrbracket_A$ be the regular language $A[v_i]$. For an expression $e \circ t$, with e derivable from E and t derivable from T in Figure 2.1, for a given assignment A , let $\llbracket e \circ t \rrbracket'_A$ be recursively defined as $\llbracket e \rrbracket_A \circ \llbracket t \rrbracket_A = \{w_l w_r \mid w_l \in \llbracket e \rrbracket_A \wedge w_r \in \llbracket t \rrbracket_A\}$.

Definition 2.2. *Subset constraint.* A constraint of the form $e \subseteq c$ is satisfiable if and only if there exists an assignment A such that $\llbracket e \rrbracket_A \subseteq \llbracket c \rrbracket_A$, i.e., A is a *satisfying assignment* for $e \subseteq c$. ⊠

Definition 2.3. *Regular Matching Assignments Problem.* Let an RMA problem instance $I = \{s_1, \dots, s_p\}$ be a set of constraints over a shared set of variables $\{v_1, \dots, v_m\}$. Each element $s_i \in I$ is a subset constraint of the form $e \subseteq c$, with e derivable from E in Figure 2.1 and c derivable from C . We say that an assignment A *satisfies* an RMA instance I if and only if A is a satisfying assignment for each $s_i \in I$. We refer to a satisfying assignment as *maximal* if, for each variable v_j in any $s_i \in I$, for any string w , $A[v_j] \cup \{w\}$ no longer satisfies I . Given a problem instance I , the RMA problem requires either (1) a satisfying assignment; or (2) a message that no satisfying assignment exists. In some cases we may, additionally, be interested in all unique satisfying assignments; in Section 2.2 we show that the number of assignments is finite. ⊠

Example 2.2. *Satisfiability and maximality.* Consider the following example over the alphabet $\{x, y\}$.

$$v_1 \subseteq L((xx)^+y) \quad v_1 \subseteq L(x^*y)$$

The correct satisfying assignment for this set of equations is $A = [v_1 \leftarrow L((xx)^+y)]$. For A to be a satisfying assignment, we require that it maps regular languages to language variables in a way that respects the input constraints. In the example, the potential solution $A' = [v_1 \leftarrow L(xy)]$ fails that test because $\llbracket v_1 \rrbracket_{A'} = L(xy) \not\subseteq L((xx)^+y)$. In addition, we may require an assignment to be *maximal*; informally that prevents assignments that do not capture enough information. In the example, the potential solution $[v_1 \leftarrow \emptyset]$ is satisfying but not maximal, because it can be extended to, for example, $[v_1 \leftarrow L(xxy)]$. \square

Example 2.3. *Multiple solutions.* RMA instances may have more than one unique satisfying assignment. For example, consider the following system:

$$\begin{aligned} v_1 &\subseteq L(x(yy)^+) \\ v_2 &\subseteq L((yy)^*z) \\ v_1 \circ v_2 &\subseteq L(xyyz|xyyyyz) \end{aligned}$$

This set of constraints has two *disjunctive* satisfying assignments:

$$\begin{aligned} A_1 &= [v_1 \leftarrow L(xyy), v_2 \leftarrow L(z|yyz)] \\ A_2 &= [v_1 \leftarrow L(x(yy|yyyy)), v_2 \leftarrow L(z)] \end{aligned}$$

Both A_1 and A_2 are *satisfying* and *maximal*. They are also inherently disjunctive, however; it is not possible to “merge” A_1 and A_2 without violating one or both properties. \square

Our definition of RMA (and the algorithm we provide to solve it) can be readily extended to support additional operations, such as union or substring indexing. For example, substring indexing might be used to restrict the language of a variable to strings of a specified length n (to model length checks in code). This could be implemented using basic operations on nondeterministic finite state

automata that are similar to the ones already implemented. Many other features, however, would make the RMA problem undecidable in general (e.g., [31]). We instead focus on a decidable theory with a provably correct core algorithm, and leave additional features for future work. In Section 2.4 we show that, without additional features, our prototype implementation can be used to solve a real-world problem.

2.2 The Concatenation-Intersection Problem

Rather than solving the general RMA problem directly, we will first consider a restricted form involving only the concatenation of two variables that each have a subset constraint. In the next subsections we will present this restricted problem, and in Section 2.3 we will use our solution to it to build a full solution for the RMA problem.

We define the *Concatenation-Intersection (CI)* problem as a subcase of the RMA problem, of the following form:

$$v_1 \subseteq c_1 \quad v_2 \subseteq c_2 \quad v_1 \circ v_2 \subseteq c_3$$

Because the form of these constraints is fixed, we define a CI problem instance strictly in terms of the constants c_1 , c_2 , and c_3 . Given three regular languages c_1 , c_2 , and c_3 , the CI problem requires the set $S = \{A_1, \dots, A_n\}$ of satisfying assignments, where each A_i is of the form $[v_1 \leftarrow x'_i, v_2 \leftarrow x''_i]$. More explicitly, we require that S satisfy the following properties:

1. **Regular:** $1 \leq i \leq n \Rightarrow A_i[v_1]$ and $A_i[v_2]$ are regular.
2. **Satisfying:** This corresponds directly to the *satisfiability* criterion for RMA:

$$\begin{aligned} 1 \leq i \leq n \Rightarrow & \llbracket v_1 \rrbracket_{A_i} \subseteq \llbracket c_1 \rrbracket \wedge \\ & \llbracket v_2 \rrbracket_{A_i} \subseteq \llbracket c_2 \rrbracket \wedge \\ & \llbracket v_1 \circ v_2 \rrbracket_{A_i} \subseteq \llbracket c_3 \rrbracket \end{aligned}$$

```

1: concat_intersect( $c_1, c_2, c_3$ ) =


---


2: Input: Machines  $M_1, M_2, M_3$  for  $c_1, c_2, c_3$ ;
3:     each machine  $M_j = \langle Q_j, \Sigma, \delta_j, s_j, f_j \rangle$ 
4: Output: Set of assignments; each  $A_i = [v_1 \leftarrow x'_i, v_2 \leftarrow x''_i]$ 
5: // Construct intermediate automata
6: let  $l_4 = c_1 \circ c_2$  s.t.  $M_4 = \langle Q_1 \cup Q_2, \Sigma, \delta_4, s_1, f_2 \rangle$ 
7: let  $l_5 = l_4 \cap c_3$  s.t.  $M_5 =$ 
8:      $\langle (Q_1 \cup Q_2) \times Q_3, \Sigma, \delta_5, s_1 s_3, f_2 f_3 \rangle$ 
9: // Enumerate solutions
10: let  $Q_{\text{lhs}} = \{f_1 q' \mid q' \in Q_3\} \subseteq Q_5$ 
11: let  $Q_{\text{rhs}} = \{s_2 q' \mid q' \in Q_3\} \subseteq Q_5$ 
12: foreach  $(q_a, q_b) \in Q_{\text{lhs}} \times Q_{\text{rhs}}$  s.t.  $q_b \in \delta_5(q_a, \epsilon)$  do
13:     let  $M'_1 = \text{induce\_from\_final}(M_5, q_1)$ 
14:     let  $M'_2 = \text{induce\_from\_start}(M_5, q_2)$ 
15:     output  $[v_1 \leftarrow M'_1, v_2 \leftarrow M'_2]$ 
16: end for

```

Figure 2.2: Constraint solving for intersection across concatenation. The algorithm relies on basic operations over NFAs: concatenation using a single ϵ -transition (line 6) and the cross-product construction for intersection (line 7-8). The two induce functions are described in the text.

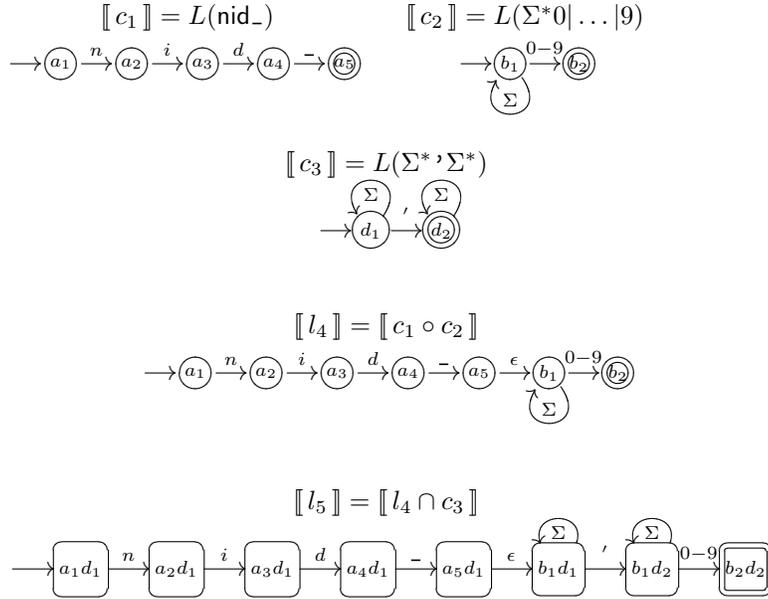


Figure 2.3: The intermediate finite state automata for the `concat_intersect` algorithm when applied to the motivating example. c_1 represents the string constant `nid_`, c_2 represents the (incorrect) input filtering on line 2, and c_3 is the language of strings that contain a single quote.

3. **All Solutions:** In addition, we want S to be nontrivial, since the *Satisfying* condition can be trivially satisfied by $S = \emptyset$:

$$\forall w \in \llbracket (c_1 \circ c_2) \cap c_3 \rrbracket, \exists 1 \leq i \leq n \text{ s.t. } w \in \llbracket v_1 \circ v_2 \rrbracket_{A_i}$$

Figure 2.2 provides high-level pseudocode for finding S . The algorithm uses operations on nondeterministic finite state automata, and we write M_i for the machine corresponding to each input language c_i or intermediate language l_i . To find S , we use the structure of the NFA M_5 (line 7–8) that recognizes $\llbracket l_5 \rrbracket = \llbracket (c_1 \circ c_2) \cap c_3 \rrbracket$. Without loss of generality, we assume that each NFA M_i has a single start state $s_i \in Q_i$ and a single final state $f_i \in Q_i$. Note that we do not assume implicit ϵ -transitions from each state to itself.

Example 2.4. *The CI algorithm.* Figure 2.3 shows the CI algorithm applied to the running example introduced in Chapter 1. The input languages c_1 and c_2 correspond to the concatenation `$newsid = "nid_" . $newsid` on line 6 of Figure 1.1, while the input c_3 corresponds to the set of strings that contain at least one quote, which is one common approximation for an unsafe SQL query [7, 32].

The machines for l_4 and l_5 in Figure 2.3 correspond to the machines constructed on lines 6 and 7–8 of Figure 2.2. The algorithm first constructs a machine for $\llbracket l_4 \rrbracket = \llbracket c_1 \circ c_2 \rrbracket$ using a single ϵ -transition between f_1 and s_2 . Next, we use the cross-product construction to create the machine that corresponds to $\llbracket l_5 \rrbracket = \llbracket l_4 \cap c_3 \rrbracket$. The set of states Q_5 for this machine corresponds to tuples in the set $(Q_1 \cup Q_2) \times Q_3$; we write $q_x q_y \in Q_5$ for the state that corresponds to $q_x \in (Q_1 \cup Q_2)$ and $q_y \in Q_3$. The transition function δ_5 is defined in the usual way.

Having constructed M_5 , we use the structure of the machine to find NFAs that represent the satisfying assignments. Intuitively, we slice up the bigger machine M_5 , which represents all solutions, into pairs of smaller machines, each of which represents a single satisfying assignment.

We are interested in those states $q_a q_b \in Q_5$ where q_a corresponds to the final state of M_1 (i.e., Q_{lhs} on line 10) or to the start state of M_2 (i.e., Q_{rhs} on line 11). Because of the way M_5 is constructed, any transitions from Q_{lhs} to Q_{rhs} must be ϵ -transitions that correspond to the original concatenation step on line 6 of Figure 2.2. For Figure 2.3, we have $Q_{\text{lhs}} = \{a_5 d_1\}$ and $Q_{\text{rhs}} = \{b_1 d_1\}$. We process each such ϵ -transition as follows:

- `induce_from_final(M_5, q_1)` (line 10) returns a copy of M_5 with q_1 marked as the only final state.
- `induce_from_start(M_5, q_2)` (line 11) returns a copy of M_5 with q_2 marked as the only start state.

We output each such solution pair. Note that, on line 15, if either M'_1 or M'_2 describe the empty language, then we reject that assignment.

The machine for l_5 in Figure 2.3 has exactly one ϵ -transition of interest. Consequently, the solution set consists of a assignment $A_1 = [v_1 \leftarrow x'_1, v_2 \leftarrow x''_1]$. x'_1 corresponds to the machine for l_5 with state a_5d_1 set as the only final state. $\llbracket x'_1 \rrbracket = L(\text{nid}_-)$, as desired. The more interesting result is x''_1 , which is the machine with start state b_1d_1 and final state b_2d_2 . The language of x''_1 captures exactly the strings that exploit the faulty safety check on line 2 of Figure 1.1: all strings that contain a single quote and end with a digit. \square

2.2.1 Correctness of the Concat-Intersect Algorithm

Having presented our high-level algorithm for solving the CI problem, we now sketch the structure of our proof of its correctness. We have formally modeled strings, state machines, our algorithm, and the desired correctness properties in version 8.1 of the Coq formal proof management system [18, 33]; proofs in Coq can be mechanically verified.

Our low-level proofs proceed by induction on the length of the strings that satisfy specific properties; we provide more detail for each individual correctness property. We say q reaches q' on s if there is a path from state q to state q' that consumes the string s .

For all regular languages c_1 , c_2 , and c_3 , if $S = \text{CI}(c_1, c_2, c_3)$ then for all elements $A_i \in S$, the following three conditions hold:

1. **Regular:** $A_i[v_1]$ and $A_i[v_2]$ are regular.

This is a type preservation property: because the operations `induce_from_final` and `induce_from_start` return NFAs, the corresponding languages are by definition regular.

2. **Satisfying:** We prove this by showing that $\forall q \in Q_{\text{lhs}}, w \in \Sigma^*, s_5$ reaches q on string $w \Rightarrow w \in c_1$, and $\forall q' \in Q_{\text{rhs}}, w \in \Sigma^*, q'$ reaches f_5 on $w \Rightarrow w \in c_2$.

3. **All Solutions:** We proceed by simultaneous induction on the structure of the machines M_3 and M_4 ; we show that $\forall w \in A_i[v_1], w' \in A_i[v_2], s_5$ reaches f_5 on $s \circ s'$ in machine M_5 (by

traversing the epsilon transition selected for A_i). Note that, since the number of ϵ -transitions in M_5 is finite, the number of disjunctive solutions must also be finite.

Our proof is available on-line¹; we believe that the presence of a mechanically checkable proof of correctness makes our algorithm attractive for use as a decision procedure or as part of a sound program analysis.

2.3 Solving General Systems of Subset Constraints

We now return to the problem of finding satisfying assignments for general regular language equations. At a high level, this requires that we generalize the `concat_intersect` algorithm from Figure 2.2. We proceed as follows:

1. To impose an ordering on the operations of our algorithm, we create a *dependency graph* based on the structure of the given equations. We describe the graph generation process in Section 2.3.1.
2. Section 2.3.2 provides a worklist algorithm that applies the `concat_intersect` procedure inductively, while accounting for repeated variable instances. That is, if a variable occurs multiple times, a solution generated for it must be consistent with all of its uses.

2.3.1 Dependency Graph Generation

Our approach to constraint solving is conceptually related to the equality DAG approach found in cooperating decision procedures [26]. Each unique language variable or constant is associated with a node in the DAG. We construct the edges of the DAG by recursively processing the regular language equation. We present the processing rules here; the algorithm in Section 2.3.2 assumes a dependency graph as its input.

Given a regular language equation, we build the dependency graph by recursive descent of the derivation under the grammar of Figure 2.1. Figure 2.4 describes the generation rules as a collecting

¹<http://www.cs.virginia.edu/~ph4u/dpr1e/proof.php>

$$\begin{array}{c}
\frac{n = \text{node}(c_i)}{\vdash c_i : n, \emptyset} T \rightarrow C \quad \frac{n = \text{node}(v_i)}{\vdash v_i : n, \emptyset} T \rightarrow V \\
\\
\frac{e_0 = t_0}{\vdash t_0 : n_0, G_0} \quad \frac{}{\vdash e_0 : n_0, G_0} E \rightarrow T \\
\\
\frac{\begin{array}{l} n_2 \text{ is fresh} \\ \vdash e : n_0, G_0 \\ \vdash t : n_1, G_1 \\ G' = \{\text{ConcatEdgePair}(n_0, n_1, n_2)\} \end{array}}{\vdash e \circ t : n_2, G_0 \cup G_1 \cup G'} E \rightarrow E \circ T \\
\\
\frac{\vdash e : n, G}{\vdash e \subseteq c : n, G \cup \{\text{SubsetEdge}(\text{node}(c), n)\}} S \rightarrow E \subseteq C
\end{array}$$

Figure 2.4: Dependency graph generation rules. We process a regular language constraint by recursive descent of its derivation; each rule corresponds to a grammar production. The `node` function returns a vertex for each unique variable or constant. For systems of multiple constraints, we take the union the dependency graphs.

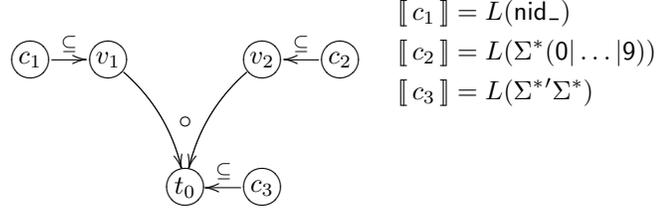


Figure 2.5: Example dependency graph. This graph corresponds to an instance of the Concatenation-Intersection problem defined in Section 2.2, using the assignments for the running SQL injection example from Section 1.2.

semantics. The rules are of the form:

$$\vdash e : n, G$$

where e is the right-hand side of a derivation step, n is the current dependency graph vertex, and G is the dependency graph. The `node` function returns a distinct vertex for each unique variable and constant. Each vertex represents a regular language; we write $\llbracket n \rrbracket$ for the language associated with vertex n . We model the graph G as a set of directed edges, of which there are two types:

- `SubsetEdge`(n_0, n_1) requires that $\llbracket n_1 \rrbracket \subseteq \llbracket n_0 \rrbracket$. In such an edge n_0 is a constant and n_1 is a language variable. We write such edges as $n_0 \rightarrow \subseteq n_1$ and refer to them as \subseteq -edges.

- $\text{ConcatEdgePair}(n_a, n_b, n_0)$ constrains the language $\llbracket n_0 \rrbracket$ to strings in $\llbracket n_a \rrbracket \circ \llbracket n_b \rrbracket$. Each constraint has two edges $n_a \rightarrow_l^\circ n_0$ and $n_b \rightarrow_r^\circ n_0$ referred to as a \circ -edge pair.

The base case rules in Figure 2.4 are those for $T \rightarrow C$ and $T \rightarrow V$. The $E \rightarrow T$ rule is the identity. All other rules extend the dependency graph through union. Note that the rule for $E \rightarrow E \circ E$ uses a fresh vertex t to represent the intermediate result of the concatenation. Finally, the top-level rule (for $S \rightarrow E \subseteq C$) adds a single \subseteq -edge from the right-hand side of the constraint c to the left-hand side.

Figure 2.5 shows the dependency graph for the example of Section 1.2. This graph corresponds an instance of the CI problem defined in Section 2.2, of the form:

$$v_1 \subseteq c_1 \quad v_2 \subseteq c_2 \quad v_1 \circ v_2 \subseteq c_3$$

Note that the edges in Figure 2.5 are strictly a description of the constraint system; they are *not* meant to indicate a strict dependence ordering. For example, changing $\llbracket c_3 \rrbracket$ to be $L(\text{nid_}'5)$ would require $\llbracket v_2 \rrbracket$ to contain only the string $'5$, even though there is no forward path through the graph from c_3 to v_2 .

2.3.2 Solving General Graphs

We now provide a general algorithm for the RMA problem defined in Section 2.1. Given a system of regular language equations, our algorithm returns the full set of disjunctive satisfying assignments from language variables to regular languages. The essence of our algorithm is the repeated application of a generalized (i.e., able to handle groups of nodes connected by concat edges) version of the `concat_intersect` algorithm.

The algorithm keeps a worklist of partially-processed dependency graphs along with mappings from vertices to finite state automata; the use of a worklist is necessary to handle disjunctive solutions. The initial worklist consists of the dependency graph that represents the regular language equation in full. The the initial node-to-NFA mapping returns Σ^* for vertices that represent a variable, and $\llbracket c_i \rrbracket$ for each constant c_i .

Figure 2.6 provides pseudocode for the algorithm, which is recursive. The algorithm consists of several high-level stages, which are applied iteratively:

1. On lines 3–8, we solve basic constraints. Many constraints can be resolved by eliminating vertices that represent constants in topological order. For example, the system

$$v_1 \subseteq c_1 \quad v_1 \subseteq c_2 \quad v_2 \subseteq c_1 \quad v_2 \subseteq c_2$$

can be processed by simply setting $A[v_2] = A[v_1] = \llbracket c_1 \cap c_2 \rrbracket$. This step does not require any calls to the `concat_intersect` procedure described in Section 2.2. Also note that this step never generates more than one set of solutions.

The `sort_acyclic_nodes` invocation on line 3 finds vertices that qualify for this treatment, and sorts them topologically. The `reduce` function performs NFA intersections (to satisfy subset constraints) and concatenations (to satisfy concatenation constraints), and removes nodes from the graph that have no further inbound constraints.

2. On lines 9–15, we apply the CI algorithm inductively to handle nodes of the graph that have both concatenation and subset constraints. We refer to connected groups of those nodes as *CI-groups*; we define such groups formally in Section 2.3.3.

Each call to `gci` (for *generalized concat-intersect*) on line 11 eliminates a single CI-group from the dependency graph, yielding separate node-to-NFA mappings for each disjunctive solution. These solutions are added to the worklist on lines 13–14.

3. Lines 16–23 determine what to do next. We either (1) terminate successfully (line 17); (2) continue to solve the current graph (line 19); (3) attempt to solve a new graph from the worklist (line 21); or (4) terminate without having found a satisfying set of inputs (line 23).

```

1: solve_dependency_graph(queue  $Q$ , node set  $S$ ) =


---


2: let  $\langle G, F \rangle : \text{graph} \times (\text{node} \rightarrow \text{NFA}) = \text{take from } Q$ 
3: let  $N : \text{node list} = \text{sort\_acyclic\_nodes}(G)$ 
4: for  $0 \leq i < \text{length}(N)$  do
5:   let  $n : \text{node} = N[i]$ 
6:   let  $\langle G', F' \rangle : \text{graph} \times (\text{node} \rightarrow \text{NFA}) = \text{reduce}(n, G, F)$ 
7:    $F \leftarrow F'; G \leftarrow G'$ 
8: end for
9: let  $C : \text{node set} = \text{find\_free\_group}(G)$ 
10: if  $|C| > 0$  then
11:   let  $\langle G', R \rangle : \text{graph} \times (\text{node} \rightarrow \text{NFA}) \text{ list} = \text{gci}(C, G, F)$ 
12:    $G \leftarrow G'; F \leftarrow \text{head}(R)$ 
13:   foreach  $r \in \text{tail}(R)$  do
14:     add  $\langle G, r \rangle$  to end of  $Q$ 
15:   end if
16: if  $\forall s \in S. F[s] \neq \emptyset \wedge |G| = 0$  then
17:   return  $F$ 
18: else if  $\forall s \in S. F[s] \neq \emptyset \wedge |G| > 0$  then
19:   return solve_dependency_graph( $\langle q, F \rangle :: Q, S$ )
20: else if  $\exists s \in S \text{ s.t. } F[s] = \emptyset \wedge |Q| > 0$  then
21:   return solve_dependency_graph( $Q, S$ )
22: else
23:   return no assignments found

```

Figure 2.6: Constraint solving algorithm for general dependency graphs over string variables. The algorithm uses a worklist of dependency graphs and node-to-NFA mappings. The graph represents the work that remains; successful termination occurs if all nodes are eliminated.

2.3.3 Solving CI-Groups Integrally

We define a *CI-group* as any set of nodes in which every node in the set is connected by a \circ -edge to another node in the set. The directions of the \circ -edges do not matter. In Figure 2.8 the nodes $\{v_a, v_b, v_c, t_1, t_2\}$ form a CI-group; we will use this example to illustrate how these groups can be solved.

The purpose of the generalized concat-intersect (*gci*) procedure referenced in Figure 2.6 is to find a solution (i.e., a mapping from variables to NFAs) for the nodes involved in a CI-group. Since an RMA problem may admit multiple disjunctive solutions (see Example 2.3), the output of *gci* is a set of such solutions.

The *gci* algorithm solves a CI-group by repeatedly processing subset constraints and concatenation constraints. In the *concat_* intersect algorithm (Figure 2.2), the final solutions M'_1 and M'_2 were both sub-NFAs of a larger NFA. Similarly, the solution for one variable in a CI-group may be a sub-NFA of the solution for another variable. A variable may appear as the operand in more than

one concatenation; in that case we must take care to find an assignment that satisfies all constraints simultaneously. The correctness of the `gci` algorithm is based on two key invariants: operation ordering and shared solution representation.

The first invariant, *operation ordering*, requires that inbound subset constraints be handled before concatenation constraints. The importance of this ordering can be seen in Figure 2.5. Initially, $\llbracket v_1 \rrbracket = \llbracket v_2 \rrbracket = L(\Sigma^*)$. If we mistakenly process the concat edge first, we obtain $\llbracket t_0 \rrbracket = L(\Sigma^*\Sigma^*)$. If we then process the subset edges, we obtain $\llbracket v_2 \rrbracket = \llbracket c_2 \rrbracket$, which is not correct — the correct solution, as described in Section 2.2, is $\llbracket v_2 \rrbracket = L(\Sigma^*\Sigma^*(0|\dots|9))$. To obtain the correct solution and “push back” subset constraints through concatenations, we must process subset constraints first.

The second invariant, *shared solution representation*, ensures that updates to the NFA $F[v]$ representing the solution for a variable v are also reflected in updates to the solutions to all variables v' that are sub-NFAs of $F[v]$. In Figure 2.5, an update to the NFA for t_0 must also be reflected in the NFAs for v_1 and v_2 . Our `gci` implementation maintains a shared pointer-based representation so that if the NFA for t_0 is changed (e.g., is subjected to the product construction to handle a subset constraint), then the NFAs for v_1 and v_2 are automatically updated.

These two invariants allow us to handle nested concatenation operations naturally. Consider the following system:

$$\begin{aligned} (v_1 \circ v_2) \circ v_3 \subseteq c_4 \quad v_1 \subseteq c_1 \\ v_2 \subseteq c_2 \quad v_3 \subseteq c_3 \end{aligned}$$

with the parentheses added for clarity. In this case, the dependency graph will be several concatenations “tall.” The final subset with $\llbracket c_4 \rrbracket$, notably, can affect any of the variables v_1 , v_2 , and v_3 . If the constraints are processed in the right order, the NFAs for v_1 , v_2 and v_3 will all be represented as sub-NFAs of a single larger NFA.

Figure 2.7 provides high-level pseudocode for solving a single CI-group. The expected output is a set of node-to-NFA mappings (one mapping for each disjunctive solution). The algorithm works as follows:

1. The nodes are processed in topological order (line 2).

```

1: gci(node set  $C$ , graph  $G$ , node  $\rightarrow$  NFA  $F$ ) =
2: let ordered : node list = topo_sort( $C$ )
3: let solution : node  $\rightarrow$  ((node  $\times$  subNFA) set) = empty
4: foreach  $n$  : node  $\in$  ordered do
5:   handle_inbound_subset_constraints( $n, G, F$ )
6:   if  $n$  has an outbound concat constraint to node  $m$  then
7:     handle_concat_constraint( $n, m, G, F$ )
8:     foreach  $n' \in C$  do
9:       foreach  $(n'', \text{states}) \in \text{solution}[n']$  do
10:        if  $n'' = n$  then
11:          update_tracking( $\text{solution}[n'], n$ )
12:        end for
13: let  $S$  : state pair set = empty
14: foreach  $m \in C$  s.t.  $\text{solution}[m]$  is empty do
15:    $S \leftarrow S \cup \text{all\_combinations}(m, F[m])$ 
16: end for
17: return generate_NFA_mappings( $\text{solution}, G, C, S$ )

```

Figure 2.7: Generalized concat-intersect algorithm for finding a set of disjunctive solutions for a group of nodes connected by \circ -edges. Inbound subset constraints are processed before concatenation constraints. The solution for a node n may be a sub-NFA S of another node m 's solution; the solution mapping tracks this (i.e., $(m, S) \in \text{solution}[n]$).

2. The solution representations for each node n are tracked in $\text{solution}[n]$. There will be multiple solution entries for a given node if that node is the operand for more than one concatenation. The set $\text{solution}[n]$ contains zero or more pairs of the form (m, S) , where m is another node and S is a sub-NFA selecting part or all of the NFA for the node m . Each such pair indicates that $\llbracket n \rrbracket$ should be constrained by the NFA for node $m \neq n$, limited to the states specified by S .
3. Each node n starts without any constraints ($\text{solution}[n]$ is empty, line 3). If $\exists(m, S) \in \text{solution}[n]$ then the solution for node n is influenced by changes to the solution for node m and we say that m *influences* n .
4. If a node m has an inbound subset constraint $c \rightarrow^{\subseteq} m$, then any nodes influenced by m have their entry in solution updated to reflect the new machine for $\llbracket m \rrbracket \cap \llbracket c \rrbracket$. This is `handle_inbound_subset_constraints` on line 5 and the updates on lines 8–11.
5. If a node m is concatenated into another node t (line 6), then any nodes influenced by m will have their entry in solution updated to map to a sub-NFA of t (this maintains the shared solution representation invariant). The mapping for $\text{solution}[m]$ is extended to include the pair

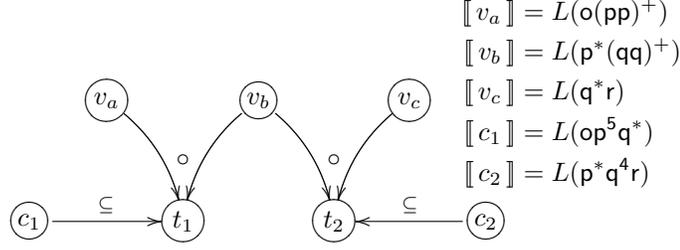


Figure 2.8: A partially-processed dependency graph that exhibits a CI-group (defined in the text). In this case, v_b is affected by both $c_1 \rightarrow^{\subseteq} t_1$ and $c_2 \rightarrow^{\subseteq} t_2$, making the two concatenations mutually dependent. The correct solution set for this graph includes all possible assignments to v_a and v_c for which there exists an assignment to v_b that simultaneously satisfies the constraints on t_1 and t_2 .

(t, S), so that m itself is now marked as influenced by t (with the appropriate subset of states S). This is `handle_concat_constraint` on line 7 and the updates on lines 8–11.

6. After all nodes have been processed (after line 12), there will be some number of non-influenced nodes that do not have any outbound o -edge pairs and are thus not in `solution`. For each such node n , the final solution is simply $F[n]$. Intuitively, these are the largest NFAs in the CI-group, since each concatenation and intersection increases the size of the resulting NFA.
7. The solution for each influenced node m (line 14) will refer to one or more non-influenced nodes, each reference coupled with an appropriate subset of states to take from the larger machine. Because of the structure for the grammar in Figure 2.1, there is always one non-influenced node; we use the NFAs of that node to generate the disjunctive solutions.

Recall that, in the `concat_intersect` procedure of Figure 2.2, we generated disjunctive solutions by selecting ϵ -transitions from the machine M_5 . To generalize this, we must generate a disjunctive solution for each *combination* of such ϵ -transitions in the non-influenced nodes' NFAs. The `all_combinations` invocation (Figure 2.7, line 15) generates these combinations, and the call to `generate_nfa_mappings` (line 17) generates a new node-to-NFA mapping for each such combination.

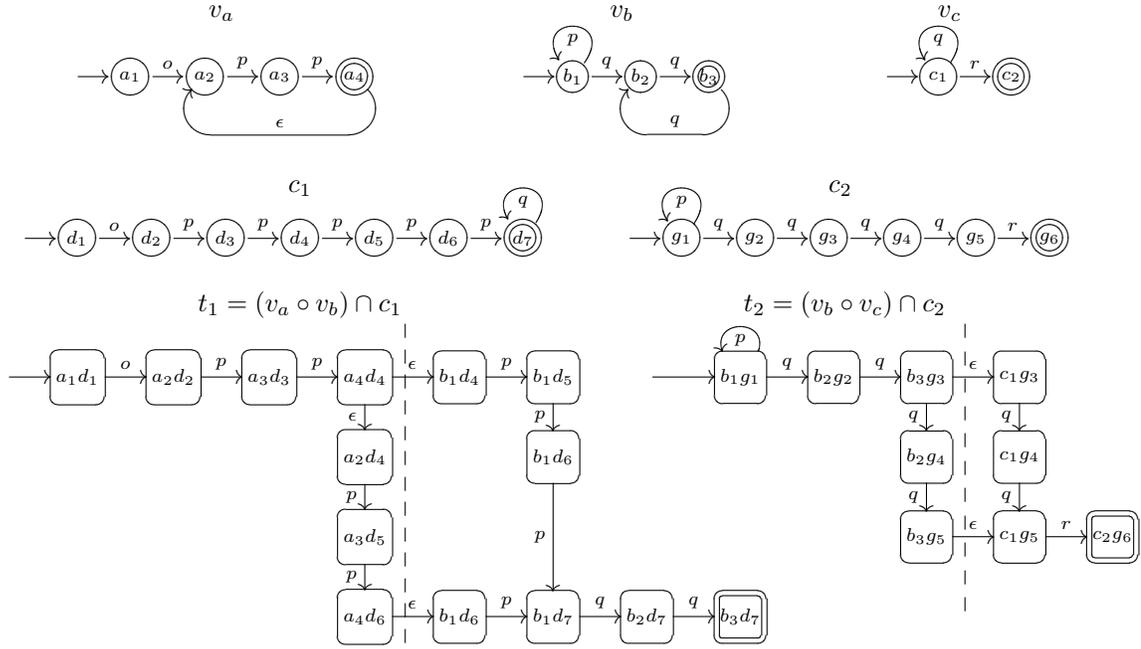


Figure 2.9: Intermediate automata for solving Figure 2.8. The `gci` procedure (Figure 2.7) finds disjunctive solutions that satisfy the constraints on t_1 and t_2 separately. It then considers all combinations of these solutions and outputs the solution combinations that have matching machines v_b .

2.3.4 Example Execution

Figure 2.9 shows the intermediate automata generated by the `gci` procedure when applied to the dependency graph of Figure 2.8. The dashed lines in the NFAs for t_1 and t_2 mark the epsilon transitions for the $v_a \circ v_b$ and $v_b \circ v_c$ concatenations, respectively. After processing each of the nodes (i.e., on line 12 of Figure 2.7), the solution mapping is as follows:

$$\begin{aligned}
 v_a &\mapsto \{(t_1, \{a_1 d_1, a_2 d_2, \dots\})\} \\
 v_b &\mapsto \{(t_1, \{b_1 d_4, b_1 d_6, \dots\}); (t_2, \{b_1 g_1, b_2 g_2, \dots\})\} \\
 v_c &\mapsto \{(t_2, \{c_1 g_3, c_1 g_4, \dots\})\} \\
 t_1 &\mapsto \emptyset \\
 t_2 &\mapsto \emptyset
 \end{aligned}$$

Note that that the machines for t_1 and t_2 each have two ϵ -transitions: t_1 connects the submachines for v_a to that of v_b , while t_2 connects submachines for v_b on the left-hand side to the submachines

for v_c on the right-hand side. This yields a total of 2×2 candidate solutions. Further, the solution mapping shows us that v_b participates in both concatenations, so for each candidate solution we must ensure that $\llbracket v_b \rrbracket$ satisfies both constraints. This leaves two satisfying assignments:

1. $A_1 = [v_a \leftarrow L(\text{op}^2), v_b \leftarrow L(\text{p}^3\text{q}^2), v_c \leftarrow L(\text{q}^2\text{r})]$

This solution used $F[t_1] \equiv L(\text{op}^5\text{q}^2)$ and $F[t_2] \equiv L(\text{p}^3\text{q}^4\text{r})$

2. $A_2 = [v_a \leftarrow L(\text{op}^4), v_b \leftarrow L(\text{pq}^2), v_c \leftarrow L(\text{q}^2\text{r})]$

This solution used $F[t_1] \equiv L(\text{op}^5\text{q}^2)$ and $F[t_2] \equiv L(\text{pq}^4\text{r})$

2.3.5 Runtime Complexity

We now turn to the runtime complexity of the decision procedure outlined in the previous section. As before, we will first discuss the `concat_intersect` procedure presented in Section 2.2, and then generalize to the full algorithm of Section 2.3.2. It should be noted that the precise complexity of combined NFA operations is the subject of ongoing research [34]. We use worst-case NFA state space complexity (i.e., the number of NFA states visited during an operation) to represent the runtime complexity of our algorithm. This is a natural representation for low-level NFA operations that require visiting all states in one or more operands, such as the cross-product construction.

We express our analysis in terms of two variables: the number of disjoint solutions for a given system, and the total number of NFA states visited. In each case, we compute a worst-case upper bound. We refer to the size $|M_i|$ of an NFA M_i as the size of its state space; let Q be an upper bound on the size of any input NFA.

The `concat_intersect` algorithm performs one concatenation operation (line 6) and then computes machine M_5 using the cross-product construction (lines 7–8). The size of the concat machine is $|M_1| + |M_2| = O(Q)$, so constructing the intersection requires visiting $|M_3|(|M_1| + |M_2|) = O(Q^2)$ states. The number of solutions in the intersection language is bounded by $|M_3|$. This is because of the way Q_{lhs} and Q_{rhs} are defined in Figure 2.2; the proof sketch in Section 2.2.1 provides more detail.

This means that the total cost of enumerating all solutions eagerly, in terms of NFA states visited, is $|M_3| \times (|M_3|(|M_1| + |M_2|)) = O(Q^3)$. We note that, in practice, we can generate the first solution without having to enumerate the others; this is why we reason separately about machine size and number of possible solutions.

The argument thus far applies to dependency graphs of the form illustrated in Figure 2.5; we now extend it to general dependency graphs. We consider two cases: (1) one or both the \circ -operands c_1 and c_2 are the result of a previous call to the `concat_intersect` procedure; and (2) the concatenation result t_0 is subject to more than one subset constraint.

In the first case, suppose v_1 is the result of a separate `concat_intersect` step; its NFA has size $O(Q^2)$ and there are $O(Q)$ possible assignments to v_1 . The machine for $v_1 \circ v_2$ then has size $O(Q^2) + O(Q) = O(Q^2)$, yielding a total enumeration size of

$$\underbrace{O(Q^2)}_{\text{solutions}} \times \underbrace{O(Q^3)}_{\text{machine size}} = O(Q^5)$$

In the second case, we consider adding an additional subset constraint to the concatenation node t_0 in the graph of Figure 2.5. Note that $|M_3|$ occurs both as a factor in the number of states visited and the number of potential solutions. We add one additional subset constraint to the concatenation (i.e., $v_1 \circ v_2 \subseteq c_4$); we assume one additional edge $c_4 \rightarrow^{\subseteq} t_0$. The enumeration then requires visiting

$$\underbrace{|M_3||M_4|}_{\text{solutions}} \times \underbrace{(|M_3||M_4|(|M_1| + |M_2|))}_{\text{machine size}} = O(Q^5)$$

states.

Informally, we note that a single `concat_intersect` call requires visiting at most Q^3 NFA states. The total cost of solving a general constraint graph grows exponentially with the number of inductive

calls to that procedure. For example, a system:

$$\begin{aligned}v_1 &\subseteq c_1 & v_1 \circ v_2 &\subseteq c_4 \\v_2 &\subseteq c_2 & v_1 \circ v_2 \circ v_3 &\subseteq c_5 \\v_3 &\subseteq c_3 & &\end{aligned}$$

requires two calls to `concat_intersect`. To enumerate the first solution for the whole system we must visit a total of $O(Q^3)$ NFA states; enumerating all possible solutions requires visiting $O(Q^5)$ states. In Section 2.4, we show that the algorithm, in spite of its exponential worst-case complexity, is efficient enough to be practical.

2.4 Evaluation

In this chapter we have defined the general Regular Matching Assignments problem for equations of regular language variables and presented a decision procedure for it. We showed how to construct a dependency graph and process parts of it in sequence to generate potentially-disjunctive solutions. The heart of our algorithm is the inductive application of our solution to the Concatenation-Intersection problem, which “pushes back” intersection constraint information through language concatenation. Having presented our algorithm and proved the correctness of its core, we now turn to an empirical evaluation of its efficiency and utility.

Recently, much attention has been devoted to static techniques that detect and report potential SQL injection vulnerabilities (e.g., [35, 36, 37]). Attacks remain prevalent [38], however, and we believe that extending static analyses to include automatically generated test inputs would make it easier for programmers to address vulnerabilities. Without testcases, defect reports often go unaddressed for longer periods of time [29, 30], and time is particularly relevant for security vulnerabilities.

To test the practical utility and scalability of our decision procedure, we implemented a prototype that automatically generates violating inputs for given SQL injection vulnerabilities. These vulnerabilities allow undesirable user-supplied commands to be passed to the back-end database of a web application. Such attacks are quite common in practice: in 2006, SQL injection vulnerabilities

Name	Version	Files	LOC	Vulnerable
eve	1.0	8	905	1
utopia	1.3.0	24	5,438	4
warp	1.2.1	44	24,365	12

Figure 2.10: Programs in the Wassermann and Su [7] data set with at least one direct defect. The *vulnerable* column lists the number of files for which we generated user inputs leading to a potential vulnerability detected by the Wassermann and Su analysis; in our experiments we attempt to find inputs for the first vulnerability in each such file.

made up 14% of reported vulnerabilities and were thus the second most commonly-reported security threat [3].

We extend an existing static analysis by Wassermann and Su [7], which detects SQL injection vulnerabilities but does *not* automatically generate testcases. Since our decision procedure works on systems of regular language equations, we constructed a basic prototype program analysis that uses symbolic execution to set up a system of string variable constraints based on paths that lead to the defect reported by Wassermann and Su. We then apply our algorithm to solve for any variables that were part of an HTTP GET or POST request.

We ran our experiments on seventeen defect reports from three programs used by Wassermann and Su [7]. The programs are large-scale PHP web applications; Figure 2.10 describes the data set in more detail. These programs were chosen because code injection defect reports were available for them via an existing program analysis; building on such an analysis helps to demonstrate the applicability of our decision procedure. More specifically, we ran our analysis on bug reports that we were able to reproduce using Wassermann and Su’s original tool and for which we could easily generate regular language constraints. Their analysis only finds *direct* defects, a term used by Wassermann and Su to refer to defects based on standard input variables, in three of their five programs; we restrict attention to three programs here.

The total program size is not directly indicative of our running time; instead, our execution time is related to the complexity of the violating path and thus of the constraints generated along it. Control flow and primitive string functions along that path contribute to the complexity of and the number of constraints and thus the complexity of the final constraint solving.

Vulnerability	FG	C	T_S
eve edit	58	29	0.32
utopia login	295	16	0.052
profile	855	16	0.006
styles	597	156	0.65
comm	994	102	0.26
warp cxapp	620	10	0.054
ax_help	610	4	0.010
usr_reg	608	10	0.53
ax_ed	630	10	0.063
cart_shop	856	31	0.17
req_redir	640	41	0.43
secure	648	81	577.0
a_cont	606	10	0.057
usr_prf	740	66	0.22
xw_mn	698	387	0.50
castvote	710	10	0.052
pay_nfo	628	10	0.18

Figure 2.11: Experimental results. For each of the SQL code injection vulnerabilities above, our tool was able to generate string values for input variables that led to the defect. $|FG|$ represents the number of basic blocks in the code; $|C|$ represent the number of constraints produced by the symbolic execution step; and T_S represents the total time spent solving constraints, in seconds.

We conducted our experiments on a 2.5 GHz Core 2 Duo machine with a 6 megabyte L2 cache and 4 gigabytes of RAM. Figure 2.11 lists our results applying our decision procedure to produce user inputs (testcases) for 17 separate reported SQL injection defects; each row corresponds to a PHP source file within the listed application. In 16 of the 17 cases, the analysis took less than one second. The `secure` testcase took multiple minutes because of the structure of the generated constraints and the size of the manipulated finite state machines. In our prototype large string constants are explicitly represented and tracked through state machine transformations. More efficient use of the intermediate NFAs (e.g., by applying NFA minimization techniques) might improve performance in those cases.

We have implemented our decision procedure as a stand-alone utility in the style of a theorem prover [16, 24] or SAT solver [22, 23]. The source code is publicly available.² Our decision procedure was able to solve all of the regular language constraints generated by our simple symbolic execution approach. The ease of constructing an analysis that could query our decision procedure, the relative efficiency of finding solutions, and the possibility of solving either part or all of the graph depending

²<http://www.cs.virginia.edu/~ph4u/dpr1e/>

on the needs of the client analysis argue strongly that our analysis could be used in practice.

2.5 Conclusion

In this chapter, we presented a decision procedure that solves systems of equations over regular language variables. We formally defined the *Regular Matching Assignments* problem (Section 2.1) and a subclass, the *Concatenation-Intersection* problem (Section 2.2). We provided algorithms for both problems, together with a mechanized, machine-checkable proof for the core `concat_intersect` procedure, which we use inductively to solve the more general RMA problem. We also outlined the space complexity of our algorithms.

We evaluated the utility and efficiency of our decision procedure empirically by generating constraints for 17 previously-reported SQL-injection vulnerabilities. In all cases, we were able to find feasible user input languages; in 16 of the 17 we were able to do so in under one second. The relative efficiency of our algorithm and the ease of adapting an existing analysis to use it suggest that our decision procedure is practical.

Chapter 3

Data Structures and Algorithms

In Chapter 2, we presented a decision procedure based on high-level automata operations, emphasizing the algorithm’s correctness and expressive utility. Recent work on string constraint solving — the previous chapter included — offers a variety of trade-offs between performance and expressiveness [39, 40, 41, 42, 43]. In this chapter, we focus on a subclass of string decision procedures that support regular expression constraints and use finite automata as their underlying representation. This includes the DRPLE implementation presented in Chapter 2, as well as the JSA [5] and Rex [42, 43] tools. These tools are implemented in different languages, they parse different subsets of common regular expression idioms, they differ in how they use automata internally, and the automata data structures themselves are different. However, each approach relies crucially on the efficiency of basic automaton operations like intersection and determinization/complementation. Existing work provides reasonable information on the relative performance of the tools as a whole, but does not give any insight into the relative efficiency of the individual data structures.

Our goal is to provide an apples-to-apples comparison between the core algorithms that underlie these solvers. To achieve this, we perform a faithful re-implementation in C# of several often-used automaton data structures. We also include several data structures that, to the best of our knowledge, have not yet featured in existing work on string decision procedures. We conduct performance experiments on an established set of string constraint benchmarks. We use both ASCII and UTF-16

encodings where possible, to investigate the impact of alphabet size on the various approaches. Our testing harness includes a relatively full-featured regular expression parser that is based on the .NET framework’s built-in parser. By fixing factors like the implementation language and the front-end parser, and by including a relatively large set of regular expression features, we aim to provide practical insight into which data structures are advisable for future string decision procedure work.

This chapter makes the following main contribution:

4. An apples-to-apples performance comparison of datastructures and algorithms for automata-based string constraint solving. We pay special attention to isolating the core operations of interest.

The rest of this chapter is structured as follows. Section 3.1 gives formal definitions of the automata constructs of interest. Section 3.2 presents the design and implementation of the data structures that we implemented. In Section 3.3, we provide experimental results using those data structures, and we conclude in Section 3.4.

3.1 Preliminaries

We assume familiarity with classical automata theory [44, 45]. In this section, we provide an alternate formalization of finite-state automata that transition based on sets of characters rather than individual symbols. In practice, this is usually how automata are implemented in code; we provide a formalization here to help formulate algorithms and their correctness arguments in the following sections.

Our representation allows multiple transitions from a source state to a target state to be “summarized” into a single *symbolic move*. Symbolic moves have labels that denote sets of characters rather than an individual characters. This representation naturally separates the structure of automata graph from the representation used for the character sets that annotate the edges. This is helpful because character sets can potentially be large; for example, the Unicode standard defines more than 1,000,000 symbols. In this chapter, we identify a general interface for character sets, and

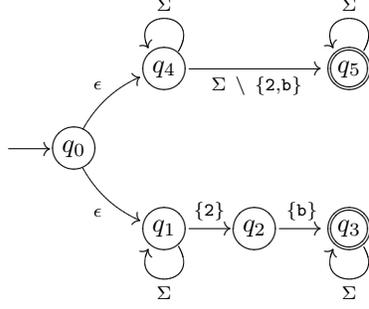


Figure 3.1: Symbolic finite-state automaton ϵ SFA generated from the regular expression $2b|[\sim 2b]$.

evaluate the performance of a number of character set implementations. The following definition builds directly on the standard definition of finite automata.

Definition 3.1. A *Symbolic Finite Automaton* (SFA) A is a tuple $(Q, \Sigma, \Delta, q^0, F)$, where Q is a finite set of *states*, Σ is the *input alphabet*, $q^0 \in Q$ is the *initial state*, $F \subseteq Q$ is the set of *final states* and $\Delta : Q \times 2^\Sigma \times Q$ is the *move relation*. ⊠

We indicate a component of an SFA A by using A as a subscript. We refer to these automata as *symbolic* based on the idea that a single move (q, ℓ, p) symbolically represents a set of transitions $[[q, \ell, p]] \stackrel{\text{def}}{=} \{(q, a, p) \mid a \in \ell\}$. We let $[[\Delta]] \stackrel{\text{def}}{=} \cup\{[[\rho]] \mid \rho \in \Delta\}$. An SFA A denotes the finite automaton $[A] \stackrel{\text{def}}{=} (Q_A, \Sigma_A, [[\Delta_A]], q_A^0, F_A)$. The *language* $L(A)$ *accepted by* A is the language $L([A])$ accepted by $[A]$.

An ϵ SFA A may in addition have moves where the label is ϵ , denoting the corresponding epsilon move in $[A]$. For $\rho = (p, \ell, q)$, let $Src(\rho) \stackrel{\text{def}}{=} p$, $Tgt(\rho) \stackrel{\text{def}}{=} q$, $Lbl(\rho) \stackrel{\text{def}}{=} \ell$ to the components of a symbolic move. We use the following notation for the set of moves starting from a given state q in A : $\Delta_A(q) \stackrel{\text{def}}{=} \{\rho \mid \rho \in \Delta_A, Src(\rho) = q\}$. In addition, we may lift functions to sets; for example, $\Delta_A(Q) \stackrel{\text{def}}{=} \cup\{\Delta_A(q) \mid q \in Q\}$. We write Δ_A^ϵ for the set of all epsilon moves in Δ_A and Δ_A^\neq for all non-epsilon moves $\Delta_A \setminus \Delta_A^\epsilon$.

An ϵ SFA A is *clean* if for all $\rho \in \Delta_A^\neq$, $Lbl(\rho) \neq \emptyset$; A is *total* if for all states $q \in Q_A$, $\Sigma_A = \bigcup_{\rho \in \Delta_A(q)} Lbl(\rho)$. As with classical automata, we can eliminate epsilon transitions (i.e., convert a ϵ SFA to an SFA).

Technique Name	Datastructure	Corresponds to
DPRLE (Chapter 2)	Single-char. hashset (OCaml; ASCII)	Eager Hashset
STRSOLVE (Chapter 4)	Char. ranges (C++; ASCII)	Lazy Range
Java String Analyzer [5]	Char. ranges (Java, Unicode)	Eager Range
Minamide [6, 7]	Single-char. functional set (OCaml; ASCII)	Eager Hashset
Veanes et al. [42]	Unary pred. (C# and Z3[16]; Unicode)	Lazy predicate
Henriksen, Jensen, et al. [46, 47]	BDDs (C)	Eager BDD

Figure 3.2: Existing automata-based string analyses, the data structures they use, and the closest-matching experimental implementation used in this chapter.

We assume a translation from regex (extended regular expression) patterns to ϵ SFAs that follows closely the standard algorithm, see e.g., [45, Section 2.5].

Example 3.1. Figure 3.1 shows a regular expression and its corresponding ϵ SFA. ☒

3.2 Automata Data structures and Algorithms

In this section, we describe the automaton data structures and algorithms of interest. We assume a graph-based data structure for the automata; each transition edge is annotated with a data structure that represents the label of that transition. Section 3.2.1 describes the data structures we use for those annotations. To put our selection of algorithms and datastructures in context, Figure 3.2 describes closely-related work together with the corresponding algorithms and datastructures presented in this chapter; Chapter 5 discusses related work in more detail. In Section 3.2.2, we discuss lazy and eager algorithms for two key operations on automata: language intersection (using the cross product construction) and language difference (using the subset construction). Later, in Section 3.3, we will evaluate experimentally how well each data structure/algorithm combination performs.

3.2.1 Representing character sets

We start by defining an interface for character set operations. This interface represents all the operations that the higher-level automata algorithms use to perform intersection and complementation. We then discuss several representations that can be used to implement this interface. The choice of a

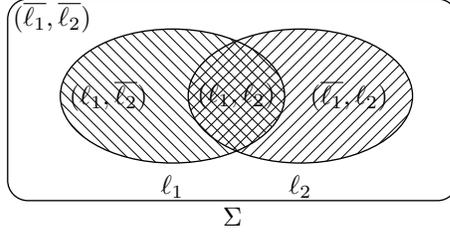


Figure 3.3: Example minterm generation for a sequence of two character sets (ℓ_1, ℓ_2) . The algorithm should lazily enumerate the nonempty areas of the Venn diagram.

representation affects the performance of the algorithms discussed below, and also how effectively the algorithms can be combined with existing solvers.

Definition 3.2. A *minterm* of a finite nonempty sequence $(\ell_i)_{i \in I}$ of nonempty subsets of Σ is a sequence $(\ell'_i)_{i \in I}$ where each ℓ'_i is ℓ_i or $\mathcal{C}(\ell_i)$ and $\bigcap_{i \in I} \ell'_i \neq \emptyset$, where $\bigcap_{i \in I} \ell'_i$ is the set *represented by* the minterm. ⊠

Intuitively, a minterm (ℓ'_1, ℓ'_2) of a sequence (ℓ_1, ℓ_2) represents a minimal nonempty region $\ell'_1 \cap \ell'_2$ of a Venn diagram for ℓ_1 and ℓ_2 ; Figure 3.3 illustrates this explicitly. Note that the *collection of all minterms* of a given sequence of nonempty sets over Σ represents a *partition* of Σ .¹

Having defined minterms and minterm generation, we are now ready to define a basic interface for character set datastructures:

Boolean operations: union (\cup), intersection (\cap), and complement (\mathcal{C}).

Emptiness checking: efficiently decide if a given set is (non)empty.

Minterm generation: compute all minterms of a sequence of sets.

For any character set representation, we can use an algorithm similar to that of Figure 3.4 to perform the minterm computation in terms of repeated complementation and intersections. We compute intersections combinatorially in rank order; once a given combination reaches the empty set at rank i , we know that derived combinations of rank $i' > i$ are necessarily empty (and thus uninteresting).

In practice, we use this algorithm for every character set representation.

¹A *partition* of Σ is a collection of nonempty subsets of Σ such that every element of Σ is in exactly one of these subsets.

```

1: minterms( $(\ell_i)_{i \in I}$  : charset seq) : (index set  $\times$  charset) seq =
2: Input: A nonempty sequence of nonempty sets  $(\ell_i)_{i \in I}$ 
3: let  $S_0 \dots S_{|I|}$  : index set set = empty
4:  $S_0 \leftarrow \{\emptyset\}$ 
5: foreach  $1 \leq r \leq |I|$  (in increasing rank order) do
6:    $S_r \leftarrow \{J \cup \{i\} \mid J \in S_{r-1}, i \in I \setminus J, \ell_i \cap \bigcap_{j \in J} \ell_j \neq \emptyset\}$ 
7:   Output each element of
8:      $\{(J, \text{concrete}((\ell_i)_{i \in I}, J)) \mid J \in S_r, \text{concrete}((\ell_i)_{i \in I}, J) \neq \emptyset\}$ 
9:   end for
10:
11:  $\text{concrete}((\ell_i)_{i \in I} : \text{charset seq}, J : \text{index set}) : \text{charset} =$ 
12: return  $(\bigcap_{j \in J} \ell_j) \cap (\bigcap_{j \in I \setminus J} \bar{\ell}_j)$ 

```

Figure 3.4: Minterm generation algorithm. The algorithm proceeds by computing minterms in rank order, outputting all nonempty combinations. Once a particular combination yields the empty set, we need not consider it for future iterations. The `minterms` function returns pairs with the indices of each minterm (first element) and the corresponding character set (second element); the second element is never empty (cf. the check on line 8). The `concrete` function returns the concrete character set indicated by a set of indices J .

For each character set representation we discuss how the above operations are supported, and indicate how the minterm generation is implemented.

Character sets as BDDs

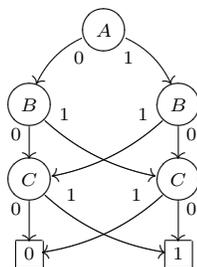
A *binary decision diagram* (BDD) is a data structure for efficiently encoding sets and relations [48, 49] as a directed acyclic graph. Given the right parameters, a BDD can represent a potentially large set of items succinctly; in addition, boolean operations over BDDs are relatively efficient. BDDs have seen use in program analysis work such as model checking [19, 50], but have not, to the best of our knowledge, been used in string constraint solving.

Example 3.2. *Binary Decision Diagram* (BDD). Consider the truth table of a three-way exclusive-or function $f(A, B, C) = A \oplus B \oplus C$, which should return 1 if an odd number of $\{A, B, C\}$ are 1:

A	B	C	$f(A, B, C)$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

The corresponding BDD considers the effect on the output of each of $\{A, B, C\}$ in a particular order.

One representation for f , assuming order (A, B, C) , is as follows:



The traversal of this BDD is analogous to the traversal a deterministic finite-state automaton over alphabet $\{0, 1\}$: we start at the topmost vertex (marked A) and traverse based on the value of the corresponding variable. If we reach one of the two *terminal* nodes, shown as squares, we output that node’s boolean value. For example, $f(0, 1, 0)$ induces the path $A \xrightarrow{0} B_{\text{left}} \xrightarrow{1} C_{\text{right}} \xrightarrow{0} 1$; note that this corresponds to the value for $f(0, 1, 0)$ in the truth table. \boxtimes

In general, a BDD represents a “compressed” version of the decision tree that corresponds to the input function. Intuitively, the worst-case size for a BDD is proportional to the size of the decision tree: exponential in the number of variables.

The general idea behind any BDD encoding is to convert set membership tests into a sequence of efficient binary decisions. For characters represented as bitvectors, this leads naturally to an encoding that tests individual bits in some order. This leaves the choice of that order, i.e. a mapping from positions in bitvectors to decision variables in the BDD representation.

Based on an informal evaluation of representation size, we use a linear ordering that considers the most significant bit (MSB) first, and the least significant bit (LSB) last. This decision is based on the observation that typical regexes make extensive use of patterns called *character classes*. Examples include $\backslash\mathbf{w}$ (matches any non-word character), $\backslash\mathbf{d}$ (matches any decimal digit), $\mathbf{p}\{\mathbf{Lu}\}$ (matches any single character in the Unicode category Lu of uppercase glyphs), etc. The class $\backslash\mathbf{w}$ is a union of seven Unicode categories, covering a set of 47,057 characters in more than 100 nonoverlapping ranges. A BDD can compactly represent those ranges while allowing for efficient set operations (e.g., set intersection corresponds to BDD conjunction).

For example the pattern $[\backslash\mathbf{w}-[\backslash\mathbf{da-d}]]$ denotes the set of all word characters that are not decimal digits or characters \mathbf{a} through \mathbf{d} , i.e., $\beta_{[\backslash\mathbf{w}-[\backslash\mathbf{da-d}]]} = \beta_{\backslash\mathbf{w}} \cap \mathbf{C}(\beta_{\backslash\mathbf{d}} \cup \beta_{\mathbf{a-d}})$. We write $\llbracket \beta \rrbracket$ for the set of characters represented by β , thus for example $\llbracket \beta_1 \cap \beta_2 \rrbracket = \llbracket \beta_1 \rrbracket \cap \llbracket \beta_2 \rrbracket$. It is easy to write infeasible

character patterns, for example $[\backslash\mathbf{w}-\backslash\mathbf{d}]$ denotes an empty set since $\llbracket\beta_{\backslash\mathbf{d}}\rrbracket \subset \llbracket\beta_{\backslash\mathbf{w}}\rrbracket$ and

$$\llbracket\beta_{\backslash\mathbf{w}} \cap \mathbb{C}(\beta_{\backslash\mathbf{d}})\rrbracket = \llbracket\mathbb{C}(\beta_{\backslash\mathbf{w}}) \cap \mathbb{C}(\mathbb{C}(\beta_{\backslash\mathbf{d}}))\rrbracket = \mathbb{C}(\llbracket\beta_{\backslash\mathbf{w}}\rrbracket) \cap \llbracket\beta_{\backslash\mathbf{d}}\rrbracket.$$

Checking the (non)emptiness of a given BDD is trivial because the empty BDD β^\perp is unique. Similarly, the BDD of all characters β^\top is unique. Except for β^\perp and β^\top , two BDDs are not guaranteed to be structurally identical by construction. They are, however, isomorphic when representing the same sets; the algorithm for checking isomorphism is linear in the size of the BDDs. We can make use of this to eliminate duplicate sets during minterm generation.

Character sets as bitvector predicates

An alternative representation for character sets is to use interval arithmetic over bitvectors or integers. We can represent those intervals symbolically in the context of a constraint solver that provides built-in support for bitvectors. We write BV^n for the sort of characters used by the solver, which is assumed to be a sort of n -bit vectors for a given fixed $n \in \{7, 8, 16\}$. Standard logic operations as well as standard arithmetic operations over BV^n , such as (unsigned) ' \leq ', are assumed to be built in. Let $\varphi_p(\chi)$ denote a predicate (with a single *fixed* free variable $\chi : BV^n$) corresponding to the regex character pattern p and let $\llbracket\varphi_p\rrbracket$ denote the set of all characters a for which $\varphi_p(a)$ is true modulo the built-in theories. For example, consider BV^7 and the character pattern $\backslash\mathbf{w}$, the predicate $\varphi_{\backslash\mathbf{w}}$ is as follows where each disjunct corresponds to a Unicode category (the Unicode categories 2, 3 and 4 are empty for the ASCII character range):

$$('A' \leq \chi \wedge \chi \leq 'Z') \vee ('a' \leq \chi \wedge \chi \leq 'z') \vee ('0' \leq \chi \wedge \chi \leq '9') \vee \chi = '-'$$

where ' \cdot ' is the bitvector representation of a character. The Boolean operations are directly supported by the corresponding built-in logical operators. For example $\llbracket\varphi_{[\backslash\mathbf{w}-\backslash\mathbf{da}-\mathbf{d}]}\rrbracket = \llbracket\varphi_{\backslash\mathbf{w}} \wedge \neg(\varphi_{\backslash\mathbf{d}} \vee \varphi_{\mathbf{a}-\mathbf{d}})\rrbracket = \llbracket\varphi_{\backslash\mathbf{w}}\rrbracket \cap \mathbb{C}(\llbracket\varphi_{\backslash\mathbf{d}}\rrbracket \cup \llbracket\varphi_{\mathbf{a}-\mathbf{d}}\rrbracket)$

For the ASCII range (or extended ASCII range), the direct range representation has several

advantages by being succinct and taking advantage of the built-in optimizations of the underlying solver. For larger ranges, like UTF-16, the representation produces predicates that scale less well. In practice, we use *if-then-else* (*Ite*) terms to encode *Shannon expansions* [51] that resemble BDDs. An *Ite*-term is a term $Ite(\psi, t_1, t_2)$ that equals t_1 , if ψ is true; equals t_2 , otherwise. Given a BDD β the corresponding predicate $Ite[\beta]$ is constructed as follows where all shared subterms are constructed only once (and cached) and are thus maximally shared in the resulting term of the solver. Given a BDD β (other than β^\perp or β^\top) we write $BitIs0(\beta)$ for the predicate over $\chi : BV^n$, that is true if and only if the i 'th bit of χ is 0, where $i = n - ordinal(\beta) - 1$ (recall that $ordinal(\beta)$ is the reverse bit position).

$$Ite[\beta] \stackrel{\text{def}}{=} \begin{cases} true, & \text{if } \beta = \beta^\top; \\ false, & \text{if } \beta = \beta^\perp; \\ Ite(BitIs0(\beta), Ite[Left(\beta)], Ite[Right(\beta)]), & \text{otherwise.} \end{cases}$$

It follows from the definition that $\llbracket \beta \rrbracket = \llbracket Ite[\beta] \rrbracket$.

Checking the nonemptiness of the set $\llbracket \varphi \rrbracket$ for a character predicate φ corresponds to checking whether the formula φ is *satisfiable* by using the constraint solver. For minterm generation we use an incremental constraint solving technique that is sometimes called *cube formula solving* [43] to enumerate concrete values.

3.2.2 Primitive automata algorithms

We will focus on two key automata algorithms: product (which corresponds to set $A \cap B$) and difference (which corresponds to $A \setminus B \equiv A \cap \overline{B}$). Viewed broadly, the majority of automaton-based string constraint solving tools use derivatives of these algorithms [5, 6, 42] to solve string constraints. Our DPRLE implementation (Chapter 2) relies on the product construction in its core solving algorithm (Section 2.2) and, in addition, uses the difference construction to allow the client to express negated constraints. Their broad use makes these algorithms good candidates for this evaluation.

The algorithms presented here assume a representation of SFAs in which labels are symbolic and use a *character set solver* that provides the functionality discussed in the previous section. Both

```

1: product(SFA A, SFA B) : SFA =


---


2: let S : stack = ( $\langle q_A^0, q_B^0 \rangle$ )
3: let V : state pair set =  $\{\langle q_A^0, q_B^0 \rangle\}$ 
4: let  $\Delta$  : move set = empty
5: while S is not empty do
6:   let  $\langle q_1, q_2 \rangle = \text{pop}(S)$ 
7:   foreach  $\rho_1 \in \Delta_A(q_1)$  and  $\rho_2 \in \Delta_B(q_2)$  do
8:     let  $\ell = \text{Lbl}(\rho_1) \cap \text{Lbl}(\rho_2)$ 
9:     if  $\ell \neq \emptyset$  then
10:      let  $p_1 = \text{Tgt}(\rho_1), p_2 = \text{Tgt}(\rho_2)$ 
11:       $\Delta \leftarrow \Delta \cup \{ (\langle q_1, q_2 \rangle, \ell, \langle p_1, p_2 \rangle) \}$ 
12:      if  $\langle p_1, p_2 \rangle \notin V$  then
13:         $V \leftarrow V \cup \{ \langle p_1, p_2 \rangle \}$ 
14:        push( $\langle p_1, p_2 \rangle, S$ )
15:      end for
16:    end while

```

Figure 3.5: Product algorithm. Constructs $A \times B$ such that $L(A \times B) = L(A) \cap L(B)$.

algorithms have a *lazy* equivalent that yields as soon as it discovers a witness. Lazy difference is *subset checking* $L(A) \subseteq L(B)$ (with witness $w \in L(A) \setminus L(B)$ iff $L(A) \subsetneq L(B)$), and lazy product is *disjointness checking* $L(A) \cap L(B) = \emptyset$ (with witness $w \in L(A) \cap L(B)$ iff $L(A) \cap L(B) \neq \emptyset$).

The product algorithm is shown in Figure 3.5. The character set solver is used for performing intersection and nonemptiness checking on labels. The difference algorithm, shown in Figure 3.6, in addition, makes use of minterm generation during an implicit determinization of its second parameter; intuitively, the algorithm is a combined product and complementation algorithm. The main advantage of the combination the two steps is the early pruning of the search stack S by keeping the resulting automaton clean.

Note that the difference algorithm reduces to complementation of B when $L(A) = \Sigma^*$, e.g., when $A = (q_A^0, \Sigma, \{q_A^0\}, \{q_A^0\}, \{(q_A^0, \Sigma, q_A^0)\})$. Then the condition $\text{Lbl}(\rho) \cap \ell_J \neq \emptyset$ above is trivially true, since $\text{Lbl}(\rho) \cap \ell_J = \ell_J$, for $\rho \in \Delta_A$. Consequently, there is no pruning of the search stack S then with respect to A , and the full complement \bar{B} is constructed. Moreover, \bar{B} is *deterministic*, since for any two distinct moves $(\langle p, Q \rangle, \ell_J, q)$ and $(\langle p, Q \rangle, \ell_{J'}, q')$ that are added to Δ above, $\ell_J \cap \ell_{J'} = \emptyset$ by definition of minterms. It follows that the difference algorithm also provides a *determinization* algorithm for B : construct \bar{B} as above and let $\text{Det}(B) = (q_B^0, \Sigma_{\bar{B}}, Q_{\bar{B}}, Q_{\bar{B}} \setminus F_{\bar{B}}, \Delta_{\bar{B}})$.

```

1: difference(SFA A, SFA B) : SFA =
2: Assume: B is total
3: let  $q^0 = \langle q_A^0, State\{q_B^0\} \rangle$ 
4: let S : stack = ( $q^0$ )
5: let V : state set = {  $q^0$  }
6: let F : state set = empty
7: let  $\Delta$  : move set = empty
8: if  $q_A^0 \in F_A \wedge q_B^0 \notin F_B$  then
9:    $F \leftarrow \{ q^0 \}$ 
10: while S is not empty do
11:   let  $\langle p, Q \rangle = \text{pop}(S)$ 
12:   let  $\{\rho_i\}_{i \in I} = \Delta_B Q$ 
13:   let  $\ell_i = Lbl(\rho_i)$  for  $i \in I$ 
14:   let  $\mathbf{M} = \text{minterms}((\ell_i)_{i \in I})$ 
15:   foreach  $\langle J, c \rangle \in \mathbf{M}$  do
16:     foreach  $\rho \in \Delta_A(p)$  s.t.  $Lbl(\rho) \cap c \neq \emptyset$  do
17:       let  $P = State\{Tgt(\rho_j) \mid j \in J\}$ 
18:       let  $q = \langle Tgt(\rho), P \rangle$ 
19:        $\Delta \leftarrow \Delta \cup \{ (\langle p, Q \rangle, Lbl(\rho) \cap c, q) \}$ 
20:       if  $q \notin V$  then
21:          $V \leftarrow V \cup \{ q \}$ 
22:         push( $q, S$ )
23:         if  $Tgt(\rho) \in F_A \wedge P \cap F_B = \emptyset$  then
24:            $F \leftarrow F \cup \{ q \}$ 
25:       end while
26: return  $A - B = (q^0, \Sigma, V, F, \Delta)$ 

```

Figure 3.6: Difference algorithm. Constructs $A - B$ such that $L(A - B) = L(A) \setminus L(B)$.

3.3 Experiments

In this section we first compare the performance of the product and difference algorithms with respect to different character set representations and eager vs. lazy versions of the algorithms. For predicate representation of character sets we use Z3 as the constraint solver. Integration with Z3 uses the .NET API that is publically available [16]. All experiments were run on a laptop with an Intel dual core T7500 2.2GHz processor. We then compare the performance of our implementation with the implementation that underlies the Java String Analyzer (JSA) [5], one of the first string analysis tools. The automata library used by the JSA is a well-established open source library; we refer to it as `dk.brics.automaton` (`brics`). We use this comparison primarily as a means of external validation — in addition to analyzing the relative performance of different algorithms and datastructures, we demonstrate that our implementations are competitive, in terms of performance over our benchmarks, with the state of the art.

	A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9
$ Q $:	36	30	31	17	11	18	4573	104	2228	42
$ \Delta^\epsilon $:	25	15	10	11	4	14	8852	99	3570	32
$ \Delta^\neq $:	36	24	28	15	11	13	148	96	524	40

Figure 3.7: Sizes of ϵ SFAs A_{i-1} constructed for regexes $\#i$ for $1 \leq i \leq 10$ in the experiments. The full regexes are shown in Figure 4.4. We assume each regex has a start anchor \sim and end anchor $\$$.

Our experiment uses a set of ten benchmark regexes that have previously been used to evaluate string constraint solving tools [42, 52]. They originate from a case study and were sourced from real code [53]. The sizes of the automata constructed from the regexes are shown in Figure 3.7.

For each pair $(A_i, A_j)_{i,j < 10}$ we conducted the following experiments to compare different character set representations, algorithmic choices, and the effect of the size of the alphabet. For product we ignored the order of the arguments due to commutativity, thus there are 55 pairs in total, and for difference there are 100 pairs in total. Figure 3.8 shows the evaluation results for the difference algorithm and the product algorithm, respectively. The times exclude the construction time of ϵ SFAs from the regexes but *include* the epsilon elimination time to convert ϵ SFAs to SFAs that is a preprocessing step in the algorithms. The total time to construct the ϵ SFAs for the 10 regexes (including parsing) was 0.33 seconds (for both UTF16 as well as ASCII). For parsing the regexes we use the built-in regex parser in .NET.

The top columns correspond to the eager vs. lazy versions of the algorithms and the secondary columns correspond to whether the result is an empty automaton or a nonempty automaton. The rows correspond to the different algorithmic choices: *BDD-X* denotes the use of the BDD based solver where X is ASCII or UTF16; *Pred-ASCII* denotes the use of predicate encoding of character sets using $Z3$ predicates over BV^7 ; *Pred-UTF16* denotes the use of predicate encoding of character sets using $Z3$ *Ite*-term encodings of BDDs over BV^{16} ; *Range-ASCII* denotes the use of hashsets of character pairs; *Hash-ASCII* denotes the use of hashsets of individual characters. We do not report results for *Range-UTF16* or *Hash-UTF16* runs because they failed the majority of instances by timing out or running out of memory. Figure 3.9 Presents our experimental data graphically across all experimental runs. It suggests that our dataset covers a considerable range of “difficulty.”

		Eager		Lazy	
		Empty	Nonempty	Empty	Nonempty
Difference	BDD-ASCII	(8).33/.007	(78)36/.008	(8).33/.006	(87)41/.001
	BDD-UTF16	(8).56/.02	(78)38/.02	(8).52/.012	(87)41/.01
	Pred-ASCII	(8)1.6/.06	(78)72/.08	(8)1.6/.06	(87)58/.02
	Pred-UTF16	(8)5.5/.12	(78)179/.24	(8)5.3/.12	(87)66/.11
	Range-ASCII	(8).9/.03	(78)67/.03	(8)1/.03	(87)44/.003
	Hash-ASCII	(8).9/.03	(78)67/.03	(8)1/.03	(87)45/.003
	brics-ASCII	(9)32/.015	(72)273/.016		
brics-UTF16	(9)39/.11	(72)341/.44			
		Eager		Lazy	
		Empty	Nonempty	Empty	Nonempty
Product	BDD-ASCII	(29)9.7/.001	(26)90/.002	(29)9.7/.001	(26)19/.001
	BDD-UTF16	(29)9.7/.001	(26)92/.003	(29)9.7/.001	(26)19/.001
	Pred-ASCII	(29)10/.003	(26)142/.003	(29)10/.004	(26)25/.007
	Pred-UTF16	(29)10/.01	(26)150/.05	(29)10/.01	(26)25/.03
	Range-ASCII	(29)10/.002	(23)16/.005	(29)10/.002	(26)69/.001
	Hash-ASCII	(29)10/.002	(23)16/.005	(29)10/.002	(26)70/.001
	brics-ASCII	(25)66/.015	(19)65/.016		
brics-UTF16	(25)66/.03	(19)70/.09			

Figure 3.8: Experimental evaluation of the SFA algorithms. Each entry in the tables has the form $(n)t/m$ where n is the *number of combinations solved*, t is the *total time* it took to solve the n instances, m is the *median*. Time is in seconds. For product, the eager experiment constructs $A_i \times A_j$ for $0 \leq i \leq j < 10$; the lazy experiment tests emptiness of $L(A_i) \cap L(A_j)$ for $0 \leq i \leq j < 10$. For difference, the eager experiment constructs $A_i - A_j$ for $0 \leq i, j < 10$; the lazy experiment tests if $L(A_i) \subseteq L(A_j)$ ($L(A_i - A_j) = \emptyset$) for $0 \leq i, j < 10$. Timeout for each instance (i, j) was 20 min.

Comparison with brics. We now briefly examine the performance of our framework relative to the open source `dk.brics.automata` library. For this comparison we first serialized the automata A_i , $i < 10$, in textual format. We serialize the automata because we want to isolate the automata algorithms and avoid differences due to, for examples, the use of different regex parsers. In the measurements we excluded the time to deserialize and to reconstruct the automata in *brics*, but included the time to add epsilon transitions, as this is effectively equivalent to epsilon elimination using the *brics* library.

The Java code responsible for the lazy difference experiment is:

```
... construct Ai, Aj, epsAi, epsAj ...

long t = System.currentTimeMillis();

boolean empty = (Ai.addEpsilons(epsAi)).subsetOf(Aj.addEpsilons(epsAj));
```

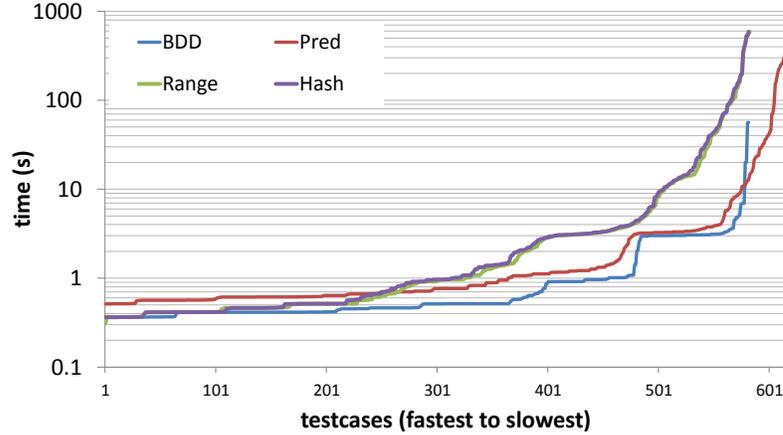


Figure 3.9: A visual representation of the experimental results across all 620 datapoints (400 for the product construction, 220 for the difference construction). The vertical axis shows the non-cumulative running time (in seconds; log scale) for each testcase. The testcases are sorted in ascending running time order *per representation*; in other words, we assume that we can predict the relative order of two inputs a priori — in practice, we found that the relative order was similar across representations. In general, the BDD representation is fastest, but fails to terminate for a small number of testcases.

```
t = System.currentTimeMillis() - t;
```

For the lazy experiment, we use the `brics.Automaton` method `subsetOf`; the eager experiment used the `brics.Automaton` method `minus`. The code for the product experiment is similar (using `intersection`). For running each instance we assigned 1.5GB to the java runtime (which was the maximum possible).

For all cases involving A_6 , the product and difference experiments timed out or exhausted memory. For the product experiment all cases involving A_8 also timed out. The instance $L(A_8) \subseteq L(A_8)$ did not time out using `brics`, while caused an out-of-memory exception using our tool during minterm generation.

For a better comparison of the “easy” cases we present the lazy results while excluding both A_6 and A_8 . Thus, there are total of 36 product instances and 64 difference instances. These results are Figure 3.10. The results show significant differences in running time across implementations. Let t_X^{op} denote the total time for experiment row X and operation op in Figure 3.10. Then:

$$\begin{aligned}
 t_{\text{brics-ASCII}}^{\text{prod}}/t_{\text{BDD-ASCII}}^{\text{prod}} &\approx 7, & t_{\text{brics-UTF16}}^{\text{prod}}/t_{\text{BDD-UTF16}}^{\text{prod}} &\approx 43, \\
 t_{\text{brics-ASCII}}^{\text{diff}}/t_{\text{BDD-ASCII}}^{\text{diff}} &\approx 1.4, & t_{\text{brics-UTF16}}^{\text{diff}}/t_{\text{BDD-UTF16}}^{\text{diff}} &\approx 21.
 \end{aligned}$$

	Product		Difference	
	Empty	Nonempty	Empty	Nonempty
BDD-ASCII	.022/.001	.024/.001	.32/.005	.33/.001
BDD-UTF16	.022/.001	.024/.001	.54/.015	.78/.002
Pred-ASCII	.07/.003	.07/.004	1.6/.06	2/.01
Pred-UTF16	.2/.01	.2/.01	5.2/.12	8.2/.05
brics-ASCII	.14/.001	.2/.015	.1/.016	.8/.016
brics-UTF16	.6/.03	1.4/.05	3.6/.05	24.8/.08

Figure 3.10: Lazy difference and product experiment times with A_6 and A_8 excluded. Each entry is *total/median* in seconds. For product, 21 instances are empty and 15 instances are nonempty. For difference, 8 instances are empty and 56 instances are nonempty.

3.4 Conclusion

In this chapter, we presented a study of automata representations for efficient intersection and difference of regular sets. We conducted this study to evaluate which combination of data structures and algorithms is most effective in the context of string constraint solving. Existing work in this area has consistently included performance comparisons at the tool level, but has been largely inconclusive regarding which automata representations work best in general. To answer this question, we re-implemented a number of data structures in the same language (C#) using a front-end parser that correctly handles a large subset of .NET’s regular expression language, and using a UTF-16 alphabet.

Our experiments showed that, over the sample inputs under consideration, the BDD-based representation provides the best performance when paired with the lazy versions of the intersection and difference algorithms. We note, however, that the performance differences across character set datastructures are less pronounced for the ASCII case and using the lazy versions of the algorithms. Overall, our results suggests that future string decision procedure work can achieve significant direct benefits by using lazy algorithms and utilizing the BDD datastructure when dealing with large alphabets. To the best of our knowledge, no existing tools currently use this combination.

Chapter 4

Solving String Constraints Lazily

The preceding chapters presented an eager decision procedure over regular languages based on automata operations (Chapter 2) and a performance comparison of automata data structures and algorithms over a synthetic benchmark (Chapter 3). At a high level, this chapter brings together some of the insights gleaned from those chapters. We present a core set of string constraints over multiple string variables and a lazy algorithm, `STRSOLVE`, for enumerating satisfying assignments. The main algorithm differs from the algorithms of Chapter 2 in that it is lazy: rather than constructing entire automata, we explore just enough of the search space to find satisfying assignments. Chapter 3 demonstrated lazy versions of automata product and difference algorithms; here we demonstrate that this approach can be extended to multivariable string constraints.

This chapter presents the following main contributions:

5. 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.
6. A comprehensive performance comparison between our prototype tool and existing implementations. We compare against CFG Analyzer [54], DPRLE (Chapter 2), Rex [42], and Hampi [39]. We use several sets of established benchmarks [39, 42]. 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.

The structure of this chapter is as follows. In Section 4.1, we provide a high-level overview of our algorithm, focusing on the (eager) construction of a graph-based representation of the search space (Section 4.1.2), followed by the (lazy) traversal of the search space (Section 4.1.3). We provide two worked examples of the algorithm in Section 4.1.4, and an informal correctness argument in Section 4.1.5. Section 4.2 provides performance results, focusing on regular language difference (Section 4.2.1), regular intersection for large strings (Section 4.2.3), and bounded context-free intersection (Section 4.2.4). we conclude in Section 4.3.

4.1 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 (cf. Chapter 2), Rex, and Hampi [42, 39], while exhibiting significantly improved average-case performance. In Section 4.1.1, we formally define the string constraints of interest. Section 4.1.2 outlines our high-level graph representation of problem instances. We then provide an algorithm for finding satisfying assignments in Section 4.1.3, and work through illustrative examples.

4.1.1 Definitions

In this chapter, we focus on a set of string constraints similar to those presented by Kiezun et al. [39], but without required a priori bounds on string variable length. In Chapter 2, we demonstrated 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 Figure 4.1. A constraint system S is a set of constraints of the form $S = \{C_1, \dots, C_n\}$, where each $C_i \in S$ is derivable from *Constraint* in Figure 4.1. *Var* denotes a finite set of string variables $\{v_1, \dots, v_m\}$. *ConstVal* denotes the set of string literals. For example, $v \in \text{ab}$ denotes that variable v must have the constant value

$Constraint$	$::=$	$StringExpr \in RegExpr$	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

Figure 4.1: 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.

ab for any satisfying assignment. We describe inclusion and non-inclusion constraints symmetrically when possible, using \diamond to represent either relation (i.e., $\diamond \in \{\in, \notin\}$).

For a given constraint system S over variables $\{v_1, \dots, v_m\}$, we write $A = [v_1 \leftarrow x_1, \dots, v_m \leftarrow x_m]$ for the *assignment* that maps variables v_1, \dots, v_m to values x_1, \dots, x_m , respectively. We define $\llbracket v_i \rrbracket_A$ to be the value of v_i under assignment A ; for a $StringExpr$ E , $\llbracket E \circ v_i \rrbracket_A = \llbracket E \rrbracket_A \circ \llbracket v_i \rrbracket_A$. For a $RegExpr$ R , $\llbracket R \rrbracket$ 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 ab^* with its language. We refer to the negation of a language using a bar (e.g., $\overline{ab^*} = \{w \mid w \notin ab^*\}$).

An assignment A for a system S over variables $\{v_1, \dots, v_m\}$ is *satisfying* iff for each constraint $C_i = E \diamond R$ in the system S , it holds that $\llbracket E \rrbracket_A \diamond \llbracket R \rrbracket$. 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, $nfa(R)$. When discussing pseudocode, we adopt the notation $nfa(R).q$ when it is necessary to refer to a particular state in $nfa(R)$ 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

```

1: follow_graph( $I$  : constraint system) =
2:   let  $G$  : directed graph = empty
3:   let  $M$  : constraint  $\rightarrow$  path = empty
4:   foreach  $C_i$  : constraint  $\in I$  do
5:     let  $(v_1 \circ \dots \circ v_n \diamond R) = C_i$ 
6:     for  $j \in 1, \dots, n - 1$  do
7:        $G \leftarrow \text{add\_edge}(G, \text{node}(v_j), \text{node}(v_{j+1}))$ 
8:        $M[C_i] \leftarrow [\text{node}(v_1), \dots, \text{node}(v_n)]$ 
9:   return ( $G, M$ )

```

Figure 4.2: 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.

automata have a single final state.

4.1.2 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 Section 4.1.3, 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 `node(v_i)`.
- For each occurrence of $\dots v_i \circ v_j \dots$ in a constraint in I , the graph has a directed edge from `node(v_i)` to `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 `[node(v_j)]`. For each constraint C_i of the form $v_k \circ \dots \circ v_m \diamond R$, we map C_i to path `[node(v_k), \dots , node(v_m)]`.

Figure 4.2 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 $[\text{node}(v_1)]$ (line 8).

Example 4.1. *Follow graph representation.* As an example, consider the following constraint system and its associated follow graph:

$$\begin{array}{l}
 C_1 = (v_1 \in \mathbf{a}^*) \\
 C_2 = (v_2 \in \mathbf{ab}) \\
 C_3 = (v_1 \circ v_2 \in \mathbf{ab})
 \end{array}
 \qquad
 \begin{array}{c}
 \vdash_{C_1} \vdash \qquad \vdash_{C_2} \vdash \\
 \textcircled{n_1} \longrightarrow \textcircled{n_2} \\
 \vdash \text{---} C_3 \text{---} \vdash
 \end{array}$$

We represent the graph G with circular vertices. The C annotations represent the domain of the mapping M . We assume $n_i = \text{node}(v_i)$. 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. \boxtimes

4.1.3 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 Section 4.1.4.

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 [55]. For constraints of the form $\dots \in R$, we construct an NFA that corresponds to $L(R)$ directly. For constraints of the form $\dots \notin R$, 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 (Section 3.2.2). For this presentation, we assume without loss that each NFA has a single final state.

```

1: type result = Unsat of result | Sat of assignment → result
2: type status = Unknown of status | StartsAt of nfastate → status
3:               | Path of nfastate → status
4: type pos = (constraint × int)
5: type searchstate = { next : var; states : var → pos → status }
6: type 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  $\forall v$  : node  $\in G$ , all_paths( $O$ .states[ $v$ ]) then
23:   return Done
24: if  $O$ .next = nil then
25:    $O$ .next  $\leftarrow$  pick_advance( $O$ ,  $G$ ,  $M$ )
26: let ( $success$ ,  $paths$ ) = advance( $O$ ,  $G$ ,  $M$ )
27: if  $\neg success$  then
28:   return Back
29: let  $O'$  : searchstate = copy( $O$ )
30:  $O'$ .next  $\leftarrow$  nil
31:  $O'$ .states[ $O$ .next]  $\leftarrow$   $paths$ 
32: foreach  $n$  : var  $\in$  succ( $O$ .next,  $G$ ) do
33:   foreach  $p = (C, i)$  : pos s.t.  $O'$ .states[ $O$ .next][ $p$ ] = Path( $x$ )  $\wedge$ 
34:          $O'$ .states[ $n$ ][ $(C, i + 1)$ ] = Unknown do
35:      $O'$ .states[ $n$ ][ $(C, i + 1)$ ]  $\leftarrow$  StartsAt(last( $x$ ))
36: return Next( $O'$ )

```

Figure 4.3: 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.

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 4.3 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).

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

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

variable v_1 occurs at *positions* (line 5) $(C_1, 1)$ 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** \rightarrow **StartsAt** \rightarrow **Unknown**.

The majority of the pseudocode in Figure 4.3 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:

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

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) performs lazy NFA intersection on all of the occurrences of variable $O.next$ to convert `StartsAt` restrictions to `Path` restrictions (or rule out that a valid path restriction exists, given the initial restrictions).
- 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 36) 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 Path restrictions.

Finding NFA Paths Based on Restrictions

The `advance` function (called on line 26 of Figure 4.3) performs all 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 other restrictions*. How we conduct the traversal for each variable depends on the restriction types for the variable's occurrences:

- An `Unknown` restriction indicates that, for the given occurrence, we do not know where the NFA path starts. Typically we can further restrict other variables to find candidate `StartsAt` restrictions for a given `Unknown` restriction. However, if the constraints are cyclic (i.e., the follow graph contains a cycle), then it may be necessary to conduct an explicit search for a start state.
- A `StartsAt` restriction requires a path through a given NFA starting at the given state; the path should agree with all other `StartsAt` and `Path` restrictions.
- A `Path` restriction requires that all other paths agree exactly with the current path.

We perform an explicit, lazy, search of the intersection (cross product) automaton. This is equivalent to a simultaneous depth-first traversal of the various automata and paths; the traversal terminates if we simultaneously reach all desired final states. In addition, we must guarantee that, given the same `searchstate`, repeated calls to `advance` yield all possible non-repeating paths through

the intersection automaton. We accomplish this by storing the search stack for NFA states between calls; if the stack is empty, we know we have exhausted all possible paths given the current constraints. Informally, the postcondition for `advance` is that any `StartsAt` restriction is replaced with a `Path` restriction, and any output `Path` restrictions agree on the concrete NFA path.

4.1.4 Worked Examples

In this subsection, we present two indicative example executions of the main solving algorithm. Example 4.2 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 Chapter 2 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 chapter (simultaneous lazy intersection of multiple automata). We discuss this example in detail with reference to line numbers in Figure 4.3.

Example 4.3 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 Figure 4.3.

Example 4.2. Consider the example constraint system, as seen before in Section 4.1.2:

$$\begin{array}{l}
 C_1 = (v_1 \in \mathbf{a}^*) \\
 C_2 = (v_2 \in \mathbf{ab}) \\
 C_3 = (v_1 \circ v_2 \in \mathbf{ab})
 \end{array}
 \qquad
 \begin{array}{c}
 \vdash_{C_1} \vdash \qquad \vdash_{C_2} \vdash \\
 \textcircled{n_1} \longrightarrow \textcircled{n_2} \\
 \vdash_{C_3} \text{-----} \vdash
 \end{array}$$

The initial searchstate (generated on line 11 of Figure 4.3) would be:

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

The main search procedure now visits this searchstate. The `visit_state` procedure, in turn, calls `pick_advance` (line 25). We assume $O.\text{next}$ 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 of 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, `paths` now equals:

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

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

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

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

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 $O.next \leftarrow v_1$, 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:

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

On the next iteration, **pick_advance** again selects v_2 . A call to **advance** yields agreeing paths from $\text{nfa}(C_2).s$ to $\text{nfa}(C_2).f$ and from $\text{nfa}(C_3).s$ to $\text{nfa}(C_3).f$. 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 4.3. 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 Figure 4.3. Consider the following constraint system, which contains a cyclic order-dependency across two variables:

$$\begin{array}{l}
C_1 = (v_1 \in \mathbf{a}^*) \\
C_2 = (v_2 \in \mathbf{b}^*) \\
C_3 = (v_1 \circ v_2 \in \mathbf{aa}(\mathbf{b})^*) \\
C_4 = (v_2 \circ v_1 \in \mathbf{bb}(\mathbf{a})^*)
\end{array}
\quad
\begin{array}{c}
\vdash_{C_1} \vdash \\
\vdash_{C_3} \vdash \\
\vdash_{C_2} \vdash \\
\vdash_{C_4} \vdash \\
\begin{array}{c}
\textcircled{n_1} \longrightarrow \textcircled{n_2} \\
\textcircled{n_2} \longrightarrow \textcircled{n_1}
\end{array}
\end{array}$$

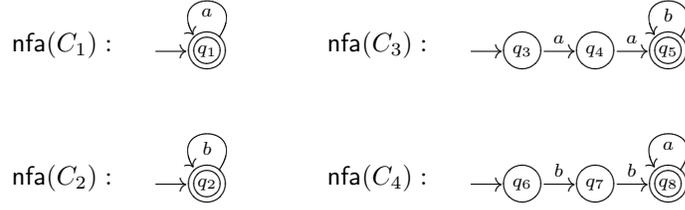
The initial search state for this constraint system looks as follows:

$$\begin{array}{l}
\{ \text{next} = \text{nil}; \\
\text{states} = \{v_1 \mapsto \{(C_1, 1) \mapsto \text{StartsAt}(\text{nfa}(C_1).s); \\
\qquad\qquad\qquad (C_3, 1) \mapsto \text{StartsAt}(\text{nfa}(C_3).s); \\
\qquad\qquad\qquad (C_4, 2) \mapsto \text{Unknown} \}; \\
v_2 \mapsto \{(C_2, 1) \mapsto \text{StartsAt}(\text{nfa}(C_2).s); \\
\qquad\qquad\qquad (C_3, 2) \mapsto \text{Unknown}; \\
\qquad\qquad\qquad (C_4, 1) \mapsto \text{StartsAt}(\text{nfa}(C_4).s); \}\}\}
\end{array}$$

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 $\text{nfa}(C_4)$ and update the v_1 state to include $(C_4, 2) \mapsto \text{StartsAt}(q)$. 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

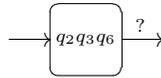
following NFAs:



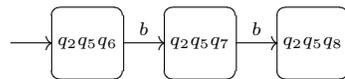
The algorithm randomly selects v_2 to restrict; this corresponds to “cutting” the $n_1 \rightarrow n_2$ edge in the follow graph. This means we need to find a start state for occurrence $(C_3, 2)$ of variable v_2 . We begin with the start state of $\text{nfa}(C_3)$: state q_3 , which yields the following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

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

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 $\text{nfa}(C_3)$, and update the search state accordingly. If we select q_5 , our search is more fruitful:

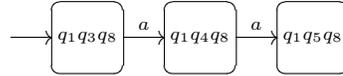


This right-most intersection state is of interest because it represents final states q_5 and q_2 for two constraints $(C_2$ and $C_3)$ in which v_2 occupies the final position. At this point, we can try to set **Path**

restrictions for v_2 and start the search for path restrictions for v_1 :

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

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 looks as follows:



This yields the following final searchstate:

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

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

4.1.5 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 Section 4.1.1 and the pseudocode of Figure 4.3 when necessary.

Definition 4.1. *Soundness:* $\forall I, \text{search}(\text{follow_graph}(I)) = \text{Sat}(A) \Rightarrow \forall (E \diamond R) \in I, \llbracket E \rrbracket_A \diamond \llbracket R \rrbracket$ \boxtimes

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 $C = (\dots \diamond R)$, if we concatenate the variable assignments, we end up with a string w that must (by construction) take $\text{nfa}(C).s$ to $\text{nfa}(C).f$, showing $w \diamond R$.

Definition 4.2. *Completeness:* $\forall I, \text{satisfiable}(I) \Rightarrow \text{search}(\text{follow_graph}(I)) \neq \text{Unsat}$ \boxtimes

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.

Definition 4.3. *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 Figure 4.3 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 a finite number of non-repeating paths 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, the `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.

4.2 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` [54], `DPRLE` (Chapter 2), `Hampi` [39], and `Rex` [42, 56]. The experiments are as follows:

- In Section 4.2.1, we re-use the benchmarks used in Chapter 3 (Figure 4.4). 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 [53]. We compare `DPRLE`, `Hampi`, `Rex`, and our `STRSOLVE` prototype, running each on all 100 pairs of regular expressions.

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

Figure 4.4: Regular expressions used for Experiment 1. The notation follows that of the .NET framework [42]; 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.

- In Section 4.2.2, we take a closer look at the performance characteristics of the Hampi implementation [39]. 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 Section 4.2.1 to answer that question.
- In Section 4.2.3, we reproduce and extend an experiment that was first used to evaluate the scalability of the Rex tool [42] 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 Section 4.2.4, 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 [54]. The task, for each implementation, is to generate strings of length 5, 10, and 12, for 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 2.8GHz Intel Core 2 Duo machine with 3.2GB 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 [57] and MiniSAT [21] included in the Hampi distribution. We use ZChaff [22] (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 [39] 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 use a version of the Rex framework (also featured in Chapter 3) under license from Microsoft; our version was released in March 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 use Rex' predicate-based representation for character sets (Pred in Chapter 3). This implementation uses an underlying solver, Z3 [16], to manipulate sets of characters. We assert that the performance of this implementation is indicative for the tool. We found it to be the second-fastest datastructure for a 7-bit alphabet (cf. Section 3.3), but it terminates for a bigger subset of the experimental data compared to the BDD-based implementation. 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.

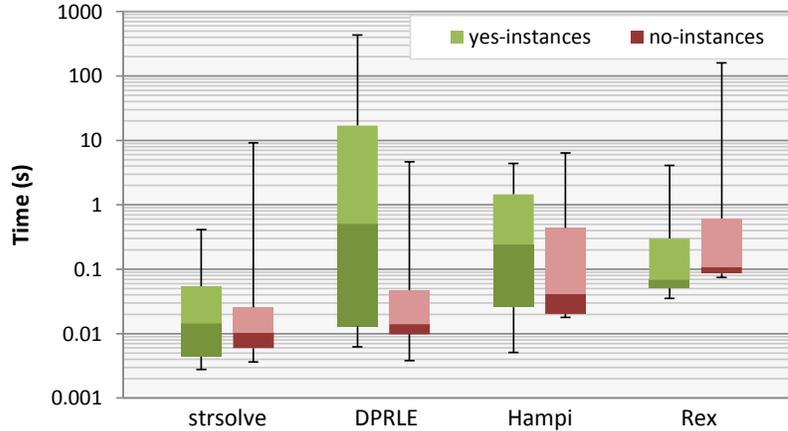


Figure 4.5: String generation time distributions (log scale), grouped by yes- and no-instances (left and right of each pair, respectively). The boxes represent the 25th through 75th percentile; the whiskers represent the 5th through 95th percentile.

4.2.1 Experiment 1: Regular Set Difference

In this experiment, we test the performance of DPRLE (Chapter 2), Hampi [39], Rex [42], and STRSOLVE on a set difference task. We reproduce an experiment originally used to test the symbolic difference construction of [42]. This experiment uses ten benchmark regular expressions presented by [53]; 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 Figure 4.4.

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 $\text{or}("", a)$ in the input language for Hampi). These expressions are presented in the format used for the Microsoft .NET framework. The *Size* column in Figure 4.4 shows the size of each regular expression after expansion. We note that there is a substantial range of sizes: from 228B (number two) to 369KB (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 as soon as Hampi finds a string; this represents a typical usage scenario¹ 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 4.5 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 $L(\bar{b})$: determinization and complementation are performed on this (potentially large) automaton lazily.

¹Hampi has since added support for ranges of length bounds; at the time of writing, it is implemented using a very similar approach.

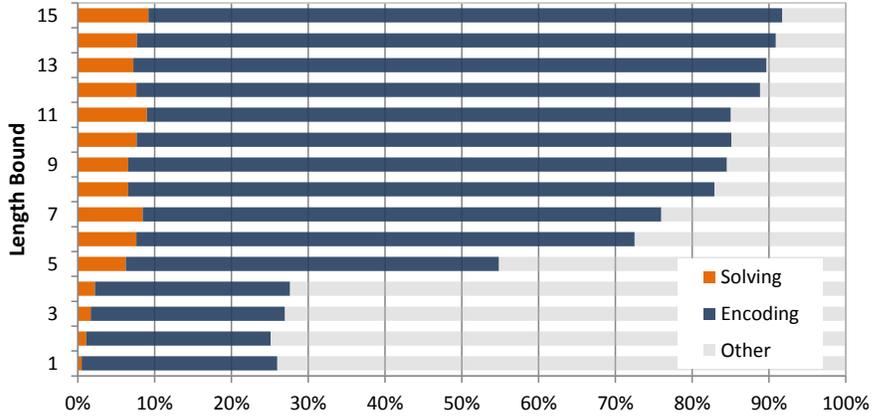


Figure 4.6: Hampi execution time breakdown for the dataset of Section 4.2.1. In this graph, **Encoding** refers to the process of converting a string constraint system into a bitvector constraint system; **Solving** refers to the time taken to solve that bitvector constraint system. We show the breakdown for length bounds $[1; 15]$; each horizontal bar represents the average of 100 samples.

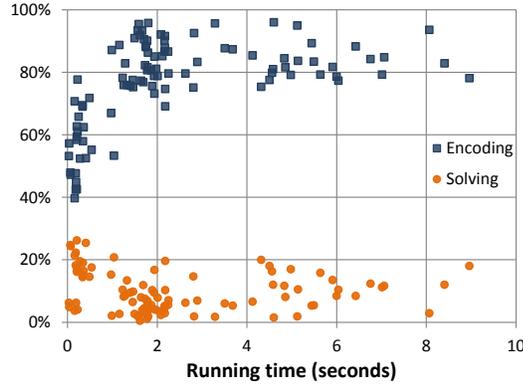


Figure 4.7: Relative time spent by Hampi on encoding and solving for the $k = 15$ case (cf. Figure 4.6); the vertical axis shows percentage of total run time, while the horizontal axis represents total solving time.

4.2.2 Experiment 2: Hampi’s Performance

We now take a closer look at the performance breakdown for the Hampi [39] 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 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 Section 4.2.1. 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 4.6 (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 4.6 (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, Figure 4.6 (right) shows that this is consistently true across test cases, not just when averaging. In fact, the $k = 15$ 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.

4.2.3 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 [42]. We compare the performance of Hampi, DPRLE, Rex, and STRSOLVE.

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

$$[a-c]^*a[a-c]^{\{n+1\}} \quad \text{and} \quad [a-c]^*b[a-c]^{\{n\}}$$

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,

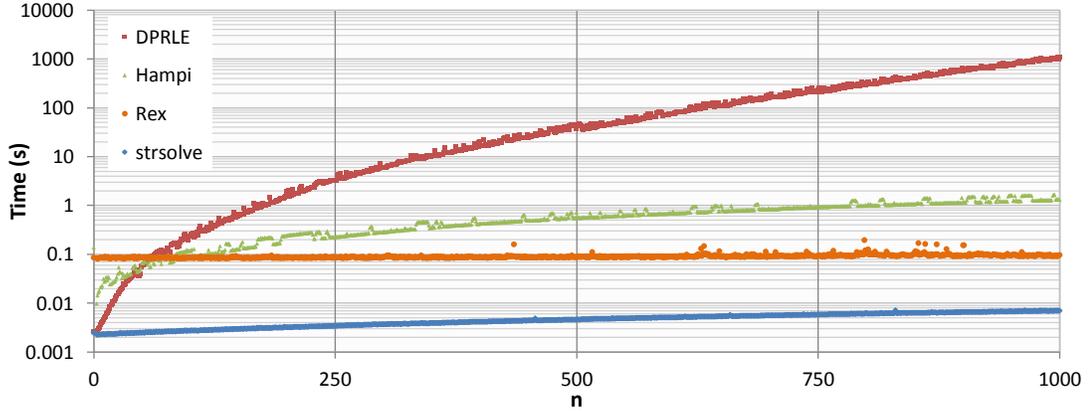


Figure 4.8: String generation times (log scale) for the intersection of the regular languages $[a-c]^*a[a-c]^{\{n+1\}}$ and $[a-c]^*b[a-c]^{\{n\}}$, for n between 1 and 1000 inclusive.

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 4.8 shows our results. Our prototype is, on average, $118\times$ faster than Hampi; the speedup ranges from $4.4\times$ to $239\times$. 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.

4.2.4 Experiment 4: Length-Bounded Context-Free Intersection

In this experiment, we compare the performance of CFG Analyzer (CFG A) [54], Hampi [39], and STRSOLVE. The experiment is similar in spirit to a previously published comparison between Hampi and CFG A: from a dataset of approximately 3000 context-free grammars published with CFG A,

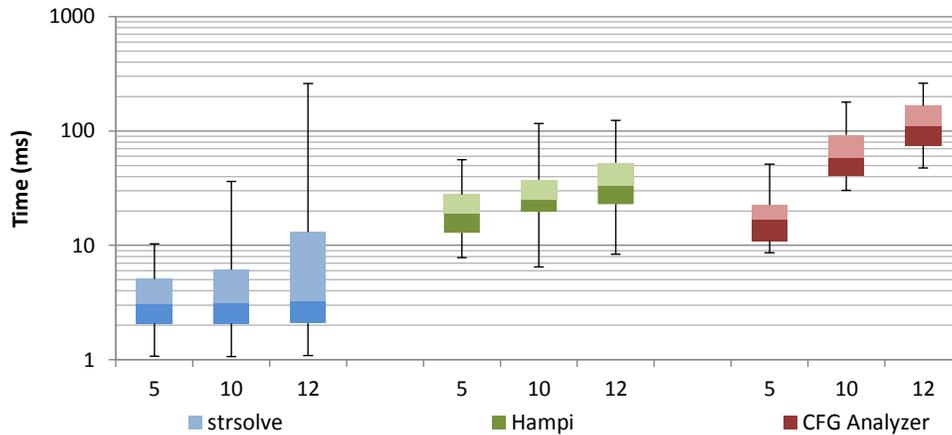


Figure 4.9: String generation times (log scale) for the intersection of context-free grammars. The grammar pairs were randomly selected from a dataset by Axelsson et al. [54]. Length bounds are 5, 10, and 12. Each column represents 85 data points; the bars show percentile 25 through 75 and the whiskers indicate percentile 5 through 95.

we randomly select pairs of grammars and have each tool search for a string in the intersection for several length bounds.

CFGGA 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. CFGGA 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 4.9 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.

4.3 Conclusion

In this chapter, we presented an extension of the lazy automata algorithms described in Chapter 3. Our algorithm constraints similar to those presented in Chapter 2, but rather than working at the level of entire regular sets, it generates single string assignments for multivariate constraints. In principle, the lazy algorithm presented in this chapter shares the worst-case complexity of the eager algorithm of Chapter 2. In practice, the lazy algorithm is designed to 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 [54], DPRLE (Chapter 2), Rex [42] (also featured in Chapter 3), and Hampi [39]. We used several sets of previously published benchmarks [39, 42]; the results show that our approach is, on average, several orders of magnitude faster than the other implementations. We believe that as string constraint solvers continue to mature, performance will be a key factor in their adoption in program analysis work. The lazy algorithm presented in this chapter is a first step in making string constraint solvers significantly outperform their alternatives, such as ad hoc implementations or the use of other constraint types (such as bitvectors, cf. Section 4.2.2) to model string constraints.

Chapter 5

Related Work

In this chapter, we discuss closely related work. This work falls into four main categories. In Section 5.1 we briefly discuss the use of decision procedures in programming languages research. In Section 5.2 we cover the original string analysis work, i.e., end-to-end program analyses with tightly coupled models for strings. Section 5.3 discusses related string constraint solvers, as well as the context of the contributions presented in Chapters 2–4. Finally, in Section 5.4, we briefly discuss work that is less closely related, but still shares theoretical or algorithmic concepts with the work presented in this dissertation.

5.1 The Use of Decision Procedures

Decision procedures have long been a fixture of program analyses. Typically a decision procedure handles queries over a certain *theory*, such as linear arithmetic, uninterpreted functions, boolean satisfiability [22, 23], pointer aliasing [14, 15], or bitwise operations and vectors [58, 57]. Nelson and Oppen presented a framework for allowing decision procedures to cooperate, forming an *automated theorem prover* to handle queries that span multiple theories and include first-order logic connectives [26]. In general, the *satisfiability modulo theories* problem is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality. A number of SMT solvers, such as CVC [17] and Z3 [16], are available.

SLAM [19] and BLAST [59] are well-known examples of program analyses that make heavy use of external decision procedures: both are software model checkers that were originally written to call upon the Simplify theorem prover [24] to compute the effects of a concrete statement on an abstract model. This process, called predicate abstraction, is typically performed using decision procedures [60] and has led to new work in automated theorem proving [61]. SLAM has also made use of an explicit alias analysis decision procedure to improve performance [62]. BLAST uses proof-generating decision procedures to certify the results of model checking [63], just as they are used by proof-carrying code to certify code safety [25].

Another recent example is the EXE project [64], which combines symbolic execution and constraint solving [65] to generate user inputs that lead to defects. EXE has special handling for bit arrays and scalar values, and our work addresses an orthogonal problem. While a decision procedure for vectors might be used to model strings, merely reasoning about indexed accesses to strings of characters would not allow a program analysis to handle the high-level regular-expression checks present in many string-using programs. Note that the Hampi project [39] (a string constraint solver, listed below) and EXE share the same underlying decision procedure (STP [57]).

5.2 End-to-End String Analysis

Early work in string analysis included a number of end-to-end analysis tools that, for example, statically detect string-related bugs in source code. These analyses deal with string manipulation, but often in a problem-specific way. One way to characterize the work in this dissertation is as an attempt to generalize the algorithmic insights gleaned from this work.

Christensen et al. first proposed a string analysis that soundly overapproximates string variables using regular languages. The Java String Analyzer (JSA) [5] uses finite automata internally to represent strings. The underlying `dk.brics.automaton` library is not presented as a constraint solving tool, but it is used in that way. The library, which is implemented in Java, represents automata using a pointer-based graph representation of node and edge objects; edges represent contiguous character ranges. It includes a deterministic automaton representation that is specialized

for matching given strings efficiently. This is not a common use case for constraint solving, since the goal there is to efficiently *find* string assignments rather than verify them. In Chapter 3, we examine the performance of the `dk.brics.automaton` library relative to other automaton representations.

Several other approaches include a built-in model of string operations; Minamide [6] and Wassermann and Su [7] rely on an ad-hoc OCaml implementation that uses that language’s built-in applicative data structures. The Saner project [66] combines a similar static component with an additional dynamic step to find real flaws in sanitizer behavior. The Wassermann and Su implementation [7] extends Minamide’s grammar-based analysis [6]. It statically models string values using context-free grammars, and detects potential database queries for which user input may change the intended syntactic structure of the query. In its original form, neither Wassermann and Su nor Minamide’s analysis can generate example inputs. In Chapter 2, we extend the Wassermann and Su analysis to generate indicative inputs for 17 SQL injection vulnerabilities.

In more recent work, Wassermann et al. show that many common string operations can be reversed using finite state transducers (FSTs) [67]. They use this method to generate inputs for SQL injection vulnerabilities in a concolic testing setup. Their algorithm is incomplete, however, and cannot be used to soundly rule out infeasible program paths. Yu et al. solve string constraints [68, 69, 70] for abstract interpretation, using approximations (“widening automata”) for non-monotonic operations, such as string replacement, to guarantee termination. Their approach has been extended to handle symbolic length constraints through the construction of length automata [71]. In general, the backward component of this component is analogous to constraint solving. At a high level, each of these techniques is an end-to-end analysis with a tightly integrated string model; this dissertation focuses on providing a string decision procedure that is useful for many client analyses.

Godefroid et al. [8] use the SAGE architecture to perform guided random input generation (similar to previous work on random testcase generation by the same authors [9, 10]). It uses a grammar specification for valid program inputs rather than generating arbitrary input strings. This allows the analysis to reach beyond the program’s input validation stages. Independent work by Majumdar and Xu [11] is similar to that of Godefroid et al.; CESE also uses symbolic execution to find inputs that

are in the language of a grammar specification. Similar projects include search-based testing [72] and CUTE [73]. These projects could potentially benefit from decision procedures for strings and regular expressions when performing symbolic execution: “Test case generation for web applications and security problems requires solving string constraints and combinations of numeric and string constraints” [74, Sec. 4]. One such example is that of the Symbolic PathFinder [75], which has been extended with a symbolic string analysis by Fujitsu and applied to the testing of web applications [76].

5.3 String Decision Procedures

The last few years have seen significant interest in string constraint solvers. The DPRLE work (Chapter 2, published at PLDI [77]) and Hampi [39] were among the first to claim standalone string constraint solvers.

The Rex tool, used for experimentation in Chapter 3 and Chapter 4, provides a SFA representation that is similar to the formal definition given in Section 3.1. The core idea, based on work by van Noord and Gerdeman [78], is to represent automaton transitions using logical predicates. Rex works in the context of *symbolic language acceptors*, which are first-order encodings of symbolic automata into the theory of algebraic datatypes. The Rex implementation uses the Z3 satisfiability modulo theories(SMT) solver [79] to solve the produced constraints. The encoding process and solving process are completely disjoint. This means that many operations, like automaton intersection, can be offloaded to the underlying solver. For example, to find a string w that matches two regular expressions, r_1 and r_2 , Rex can simply assert the existence of w , generate symbolic automaton encodings for r_1 and r_2 , and assert that s is accepted by both those automata. We refer to this as a *Boolean encoding* of the string constraints.

The initial Rex work [42] explores various optimizations, such as minimizing the symbolic automata prior to encoding them. These optimizations make use of the underlying SMT solver to find combinations of edges that have internally-consistent move conditions. Subsequent work [43] explored the trade-off between the Boolean encoding and the use of automata-specific algorithms for language intersection and language difference. In this case, the automata-specific algorithms make

repeated calls to Z3 to solve *cube formulae* to enumerate edges with co-satisfiable constraints. In practice, this approach is not consistently faster than the Boolean encoding. We use the high-level framework provided by Rex to implement the experiments of Chapter 3; this work was published at VMCAI [56].

In Chapter 4, we present a lazy backtracking search algorithm for solving regular inclusion constraints. An earlier version of this work was published at ASE [52]. The underlying automaton representation, written in C++, is based on the Boost Graph Library [80] and allows for a variety of adjacency list representations. We annotate transitions with integer ranges using the off-the-shelf Boost interval container library. The implementation pays special attention to memory management, using fast pool allocation for small objects such as the abstract syntax tree for regular expressions. This implementation uses lazy intersection and determinization algorithms, allowing for significant performance benefits relative to many other implementations (cf. Section 4.2).

Bjørner et al. describe a set of string constraints based on common string library functions [41]. The approach is based on a direct encoding to a combination of theories provided by the Z3 SMT solver [79]. In addition, they show that the addition of a replace function makes the theory undecidable. Supported operations include substring extraction (and, in general, the combination of integer constraints with string constraints), but the work does not provide an explicit encoding for regular sets.

The Hampi tool [39] uses an eager bitvector encoding from regular expressions to bitvector logic. The encoding does not use quantification, and requires the enumeration of all positional shifts for every subexpression. We provide several performance comparisons that include the Hampi implementation in Section 4.2, and we include a detailed Hampi-specific performance breakdown in Section 4.2.2.

Saxena et al. provide an extension to Hampi that supports multiple variables and length constraints [40]. The associated tool, Kaluza, builds on Hampi’s underlying solver (STP [57]) to iteratively solve integer constraints (by calling STP directly) and string constraints (using Hampi).

The CFG Analyzer tool [54] 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. We include CFG Analyzer in a performance comparison in Chapter 4.

The recent BEK project examines the use of *symbolic finite state transducers* [27, 28] as a model for string-manipulating code. Unlike traditional string analysis work, which aims to model general-purpose code by approximation, we instead model 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). In contrast, this dissertation focuses on constraint solving techniques over strings that are largely independent of programming language specifics.

5.4 Other Domains

The following work is not directly related to program analysis tools that involve strings, but it is discussed here because of theoretical ties to string constraint solving or because they explore highly optimized automata operations in some unrelated domain.

The MONA implementation [46] provides decision procedures for several varieties of monadic second-order logic (M2L). MONA relies on a highly-optimized BDD-based representation for deterministic automata. The implementation has seen extensive engineering effort; many of the optimizations are described in a separate paper [81]. It should be noted that the BDD representation discussed in Chapter 3 is distinct from the BDD representation used by the Mona tool [82, Sec. 5]. We are not aware of any work that investigates the use of multi-terminal BDDs for nondeterministic finite automata directly. We believe this may be a promising approach, although the use of BDDs complicates common algorithms like automaton minimization. In this dissertation, we restrict our attention to a class of graph-based automata representations.

There has been extensive theoretical work on language equations; Kunc provides an overview [83]. Work in this area has typically focused on complexity bounds and decidability results. Bala [84]

defines the *Regular Language Matching (RLM)* problem, a generalization of the Regular Matching Assignments (RMA) problem presented in Chapter 2 that allows both subset and superset constraints. Bala uses a construct called the *R-profile automaton* to show that solving RLM requires exponential space.

Other work explores the use of automata for arithmetic constraint solving; Boigelot and Wopler provide an overview [85]. A subset of work on explicit state space model checking (e.g., [86]) has focused on general algorithms to perform fast state space reduction; one example includes a shared set representation that trades off higher construction costs for constant-time set equality checks. Aspects of this type of work may be of future interest for automata-based string constraint solvers.

Chapter 6

Conclusion

Many program analyses and testing frameworks deal with code that manipulates string variables using high-level operations like regular expression matching and concatenation. These operations are difficult to reason about because they are not well-supported by the existing analysis machinery. We investigated the thesis:

It is possible to construct a practical algorithm that decides the satisfiability of constraints that cover both string and integer index operations, scales up to real-world program analysis problems, and admits a machine-checkable proof of correctness.

Based on the results presented in the preceding chapters, we claim that this is possible. We defined a formal theory of string constraints together with several solving algorithms. The definition represents a trade-off: it is different enough from existing theories to permit a specialized solving algorithm (cf. Section 4.2.2 for a comparison with an encoding from string constraints into bitvector constraints), but it is sufficiently similar to be used in standard contexts (e.g., symbolic execution, as demonstrated in Section 2.4).

We claimed the following six main contributions:

1. In Section 2.1, we identified and formally defined the *Regular Matching Assignments* (RMA) problem, which describes a theory of language-level constraints.

2. We exhibited an automata-based algorithm, `concat_intersect`, in Section 2.2. By virtue of being based on automaton operations, the algorithm permits a full (from-first-principles) proof of correctness proof rendered in the calculus of inductive constructions [18]. In addition, we provide an open source implementation, DPRLE.¹
3. We evaluated the expressive utility of the DPRLE implementation (2) in the context of generating testcases that trigger SQL injection vulnerabilities in a corpus of real-world PHP code (Section 2.4). We found that our tool was able to handle constraints generated using a standard symbolic execution technique.
4. We conducted an apples-to-apples performance comparison of datastructures and algorithms for automata-based string constraint solving (Chapter 3). We paid special attention to isolating the core operations of interest, and used the popular open source `dk.brics.automaton` library as a performance baseline.

We found that a BDD-based character set representation outperformed many commonly-used representations, in particular for larger alphabets such as UTF-16. In addition, we note that lazy automata algorithms are consistently more efficient than their eager counterparts.

5. Based on the insights of (4), we developed a decision procedure that supports the efficient and lazy analysis of string constraints over multiple variables (Chapter 4).

This required redefining the constraint systems formalized in (1) in terms of individual string assignments (rather than regular language assignments; Section 4.1.1). We provide an open source implementation, STRSOLVE.²

6. We conducted a comprehensive performance comparison between our STRSOLVE prototype (5) and implementations from related work (Section 4.2). We found 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.

¹<http://www.cs.virginia.edu/~ph4u/dprle/>

²<http://www.cs.virginia.edu/~ph4u/strsolve/>

These results represent some of the first concrete steps to enabling program analysis tools that can rely on off-the-shelf tools for string-related reasoning. We anticipate that, in order to see widespread use in programming languages research, string decision procedures would need to be offered as part of a multi-theory SMT framework like CVC3 [17] or Z3 [16]. Future work might reasonably focus on the integration of a theory of strings in such a framework; this poses both nontrivial theoretical challenges as well as substantial engineering challenges. We are aware of at least one tentative proposal for the standardization of a theory of string constraints as part of the SMT-LIB standard [87]. Speaking more generally, we believe there is promise in using language-theoretic constructs beyond automata for the analysis of code and code-related artifacts. For the BEK project, we used an extension of finite-state transducers to model code directly, focusing primarily on low-level string sanitization code [27, 28]. The use of transducers imposes restrictions on the type of code that can be modeled directly, but allows for a relatively rich set of analysis operations. We believe that alternate versions of this trade-off are worth exploring.

Appendix A

Library concatintersect

Below, we include the proof text for the concat intersect algorithm of Section 2.2. It defines characters, strings, nondeterministic automata, and reachability at a relatively low level in Coq [18]. We exclude the tactics for the Lemma and Theorem blocks; the full proof transcript is available on-line.¹

```
Require Import Bool. Require Import EqNat. Require Import List. Require Import Omega.
Require Import Peano_dec.
```

```
Inductive Char : Set := Char_Index : nat → Char.
```

```
Fixpoint char_eq (a b : Char) { struct b } : bool :=
  match a, b with
  | Char_Index m, Char_Index n ⇒ EqNat.beq_nat m n
  end.
```

Lemma *char_eq_dec* : $\forall a b : Char, \{a = b\} + \{a \neq b\}$.

```
Inductive String : Set :=
  | EmptyString : String
  | Str : Char → String → String.
```

```
Fixpoint concat (s s' : String) { struct s } : String :=
  match s with
  | EmptyString ⇒ s'
  | Str c w ⇒ Str c (concat w s')
  end.
```

Lemma *concat_empty* : $\forall (s : String), \text{concat } s \text{ EmptyString} = s$.

```
Inductive Symbol : Set :=
  | Character : Char → Symbol
  | Epsilon : Symbol.
```

```
Fixpoint sym_eq (a b : Symbol) { struct b } : bool :=
  match a, b with
  | Character m, Character n ⇒ char_eq m n
  | Epsilon, Epsilon ⇒ true
  | -, - ⇒ false
  end.
```

Lemma *sym_eq_dec* : $\forall a b : Symbol, \{a = b\} + \{a \neq b\}$.

Lemma *nat_neq_bneq* : $\forall n n', n \neq n' \rightarrow \text{false} = \text{beq_nat } n \ n'$.

Module Type *NFADEF*.

```
Parameter Qtype : Set. Parameter Q_dec :  $\forall a b : Qtype,$ 
```

¹<http://www.cs.virginia.edu/~ph4u/dpr1e/proof.php>

$\{a = b\} + \{a \neq b\}$. Parameter $Q : list\ Qtype$. Parameter $s : Qtype$. Parameter $f : Qtype$. Parameter $D : Qtype \rightarrow Symbol \rightarrow list\ Qtype$. Parameter $D_dec : \forall (q\ q' : Qtype) (s : Symbol),$
 $\{In\ q' (D\ q\ s)\} + \{\sim(In\ q' (D\ q\ s))\}$.

Parameter $beq_q : Qtype \rightarrow Qtype \rightarrow bool$.
Parameter $beq_q_eq : \forall q\ q' : Qtype,$
 $q = q' \leftrightarrow true = beq_q\ q\ q'$.
Parameter $beq_q_refl : \forall q : Qtype,$
 $true = beq_q\ q\ q$.

End *NFADEF*.

Module *NFA* ($n : NFADEF$).

Definition $Qtype := n.Qtype$.
Definition $Q := n.Q$.
Definition $s := n.s$.
Definition $f := n.f$.
Definition $D := n.D$.
Definition $Q_dec := n.Q_dec$.
Definition $D_dec := n.D_dec$.
Definition $beq_q := n.beq_q$.
Definition $beq_q_eq := n.beq_q_eq$.
Definition $beq_q_refl := n.beq_q_refl$.

Inductive $reachable : Qtype \rightarrow String \rightarrow Qtype \rightarrow Prop :=$
| $Done : \forall (q : Qtype), In\ q\ Q \rightarrow reachable\ q\ EmptyString\ q$
| $Step : \forall (q : Qtype) (c : Char) (s : String) (q' q'' : Qtype),$
 $In\ q\ Q \rightarrow In\ q' Q \rightarrow In\ q'' Q \rightarrow$
 $In\ q' (D\ q\ (Character\ c)) \rightarrow$
 $reachable\ q' s\ q'' \rightarrow$
 $reachable\ q\ (Str\ c\ s)\ q''$
| $StepEpsilon : \forall (q : Qtype) (s : String) (q' q'' : Qtype),$
 $In\ q\ Q \rightarrow In\ q' Q \rightarrow In\ q'' Q \rightarrow$
 $In\ q' (D\ q\ Epsilon) \rightarrow$
 $reachable\ q' s\ q'' \rightarrow$
 $reachable\ q\ s\ q''$.

Definition $language := fun\ x \Rightarrow reachable\ s\ x\ f$.

Lemma $concat_reachability : \forall (q\ q' q'' : Qtype) (s\ s' : String),$
 $reachable\ q\ s\ q' \rightarrow$
 $reachable\ q' s' q'' \rightarrow$
 $reachable\ q\ (concat\ s\ s')\ q''$.

End *NFA*.

Module $M1' ; NFADEF$.

Inductive $Qtype' : Set := Index : nat \rightarrow Qtype'$.
Definition $Qtype := Qtype'$.
Lemma $Q_dec : \forall (q\ q' : Qtype),$
 $\{q = q'\} + \{q \neq q'\}$.

Variable $s : Qtype$.
Variable $f : Qtype$.
Variable $Q' : list\ Qtype$.
Definition $Q := s :: f :: Q' : list\ Qtype$.
Variable $D : Qtype \rightarrow Symbol \rightarrow list\ Qtype$.

```

Lemma D_dec :  $\forall (q\ q' : Qtype) (s : Symbol),$ 
                $\{In\ q' (D\ q\ s)\} + \{\sim(In\ q' (D\ q\ s))\}$ .

Fixpoint beq_q (q q' : Qtype) { struct q' } : bool :=
  match q, q' with
  | Index m, Index n  $\Rightarrow$  EqNat.beq_nat m n
  end.

Lemma beq_q_eq :  $\forall q\ q' : Qtype, q = q' \leftrightarrow true = beq\_q\ q\ q'$ .

Lemma beq_q_refl :  $\forall q : Qtype, true = beq\_q\ q\ q$ .

End M1'.

Module M2'  $\{;$  NFADEF.
  Inductive Qtype' : Set := Index : nat  $\rightarrow$  Qtype'.
  Definition Qtype := Qtype'.
  Lemma Q_dec :  $\forall (q\ q' : Qtype),$ 
                 $\{q = q'\} + \{q \neq q'\}$ .

  Variable s : Qtype.
  Variable f : Qtype.
  Variable Q' : list Qtype.
  Definition Q := s :: f :: Q' : list Qtype.
  Variable D : Qtype  $\rightarrow$  Symbol  $\rightarrow$  list Qtype.

  Lemma D_dec :  $\forall (q\ q' : Qtype) (s : Symbol),$ 
                 $\{In\ q' (D\ q\ s)\} + \{\sim(In\ q' (D\ q\ s))\}$ .

  Fixpoint beq_q (q q' : Qtype) { struct q' } : bool :=
    match q, q' with
    | Index m, Index n  $\Rightarrow$  EqNat.beq_nat m n
    end.

  Lemma beq_q_eq :  $\forall q\ q' : Qtype, q = q' \leftrightarrow true = beq\_q\ q\ q'$ .

  Lemma beq_q_refl :  $\forall q : Qtype, true = beq\_q\ q\ q$ .

End M2'.

Module M3'  $\{;$  NFADEF.
  Inductive Qtype' : Set := Index : nat  $\rightarrow$  Qtype'.
  Definition Qtype := Qtype'.
  Lemma Q_dec :  $\forall (q\ q' : Qtype),$ 
                 $\{q = q'\} + \{q \neq q'\}$ .

  Variable s : Qtype.
  Variable f : Qtype.
  Variable Q' : list Qtype.
  Definition Q := s :: f :: Q' : list Qtype.
  Variable D : Qtype  $\rightarrow$  Symbol  $\rightarrow$  list Qtype.

  Lemma D_dec :  $\forall (q\ q' : Qtype) (s : Symbol),$ 
                 $\{In\ q' (D\ q\ s)\} + \{\sim(In\ q' (D\ q\ s))\}$ .

  Fixpoint beq_q (q q' : Qtype) { struct q' } : bool :=
    match q, q' with
    | Index m, Index n  $\Rightarrow$  EqNat.beq_nat m n
    end.

  Lemma beq_q_eq :  $\forall q\ q' : Qtype, q = q' \leftrightarrow true = beq\_q\ q\ q'$ .

  Lemma beq_q_refl :  $\forall q : Qtype, true = beq\_q\ q\ q$ .

End M3'.

```

Module $M1 := NFA\ M1'$. Module $M2 := NFA\ M2'$. Module $M3 := NFA\ M3'$.

Inductive $Q4 : Set :=$

| $InQ1 : M1.Qtype \rightarrow Q4$
| $InQ2 : M2.Qtype \rightarrow Q4$.

Module $M4'$; *NFADEF*.

Definition $Qtype := Q4$.

Lemma $Q_dec : \forall a\ b : Qtype,$
 $\{a = b\} + \{a \neq b\}$.

Definition $Q := (List.map\ (\text{fun}\ x \Rightarrow InQ1\ x)\ M1.Q) ++$
 $(List.map\ (\text{fun}\ x \Rightarrow InQ2\ x)\ M2.Q)$.

Definition $s := InQ1\ M1.s$.

Definition $f := InQ2\ M2.f$.

Definition $D := \text{fun}\ (q : Q4)\ (s : Symbol) \Rightarrow$
 $\text{match}\ q\ \text{with}$
| $InQ1\ q' \Rightarrow \text{if}\ \text{andb}\ (M1.beq_q\ M1.f\ q')\ (sym_eq\ s\ Epsilon)\ \text{then}$
 $(InQ2\ M2.s :: (List.map\ (\text{fun}\ x \Rightarrow InQ1\ x)\ (M1.D\ q'\ s)))$
 else
 $(List.map\ (\text{fun}\ x \Rightarrow InQ1\ x)\ (M1.D\ q'\ s))$
| $InQ2\ q' \Rightarrow (List.map\ (\text{fun}\ x \Rightarrow InQ2\ x)\ (M2.D\ q'\ s))$
 end .

Lemma $D_dec : \forall (q\ q' : Qtype)\ (s : Symbol),$
 $\{In\ q'\ (D\ q\ s)\} + \{\sim(In\ q'\ (D\ q\ s))\}$.

Fixpoint $beq_q\ (q\ q' : Qtype)\ \{\text{struct}\ q'\} : bool :=$
 $\text{match}\ q,\ q'\ \text{with}$
| $InQ1\ -, InQ2\ - \Rightarrow false$
| $InQ2\ -, InQ1\ - \Rightarrow false$
| $InQ1\ q,\ InQ1\ q' \Rightarrow M1.beq_q\ q\ q'$
| $InQ2\ q,\ InQ2\ q' \Rightarrow M2.beq_q\ q\ q'$
 end .

Lemma $beq_q_eq : \forall q\ q' : Qtype,\ q = q' \leftrightarrow true = beq_q\ q\ q'$.

Lemma $beq_q_refl : \forall q : Qtype,\ true = beq_q\ q\ q$.

End $M4'$.

Module $M4 := NFA\ M4'$.

Lemma $m4_q_union_left : \forall (q : M1.Qtype), (In\ q\ M1.Q) \rightarrow (In\ (InQ1\ q)\ M4.Q)$.

Lemma $m4_d_left : \forall (q\ q' : M1.Qtype)\ (c : Symbol), In\ q'\ (M1.D\ q\ c) \rightarrow In\ (InQ1\ q')\ (M4.D\ (InQ1\ q)\ c)$.

Lemma $m4_d_left2 : \forall (q\ q' : M1.Qtype)\ (c : Symbol), In\ (InQ1\ q')\ (M4.D\ (InQ1\ q)\ c) \rightarrow In\ q'\ (M1.D\ q\ c)$.

Lemma $m4_q_union_right : \forall (q : M2.Qtype), (In\ q\ M2.Q) \rightarrow (In\ (InQ2\ q)\ M4.Q)$.

Lemma $m4_d_right : \forall (q\ q' : M2.Qtype)\ (c : Symbol), In\ q'\ (M2.D\ q\ c) \rightarrow In\ (InQ2\ q')\ (M4.D\ (InQ2\ q)\ c)$.

Lemma $m4_d_right2 : \forall (q\ q' : M2.Qtype)\ (c : Symbol), In\ (InQ2\ q')\ (M4.D\ (InQ2\ q)\ c) \rightarrow In\ q'\ (M2.D\ q\ c)$.

Lemma $m4_q_union : \forall (q : M4.Qtype), (In\ q\ M4.Q) \rightarrow$

$\text{match}\ q\ \text{with}$
| $InQ1\ q1 \Rightarrow In\ q1\ M1.Q$
| $InQ2\ q2 \Rightarrow In\ q2\ M2.Q$

end.

Lemma *m4_no_going_back* : $\forall (q : M2.Qtype) (q' : M1.Qtype) (c : Symbol), In (InQ1 q') (M4.D (InQ2 q) c) \rightarrow False$.

Lemma *m4_left* : $\forall (q q' : M1.Qtype) (s : String), M1.reachable q s q' \rightarrow M4.reachable (InQ1 q) s (InQ1 q')$.

Lemma *m4_left_right* : $\forall (q q' : M4.Qtype) (s : String), M4.reachable q s q' \rightarrow$
 match *q, q'* with
 | *InQ1 q1, InQ1 q1'* $\Rightarrow M1.reachable q1 s q1'$
 | *InQ2 q2, InQ2 q2'* $\Rightarrow M2.reachable q2 s q2'$
 | *InQ2 -, InQ1 -* $\Rightarrow False$
 | *InQ1 -, InQ2 -* $\Rightarrow True$
end.

Lemma *m4_right* : $\forall (q q' : M2.Qtype) (s : String), M2.reachable q s q' \rightarrow M4.reachable (InQ2 q) s (InQ2 q')$.

Definition *Q5* := *prod M4.Qtype M3.Qtype*.

Module *M5'* ; *NFADEF*.

Definition *Qtype* := *Q5*.

Lemma *Q_dec* : $\forall a b : Qtype,$
 $\{a = b\} + \{a \neq b\}$.

Definition *Q* := *List.list_prod (M4.Q) (M3.Q)*.

Definition *s* := *(M4.s, M3.s)*.

Definition *f* := *(M4.f, M3.f)*.

Definition *D* := fun (*q* : *Q5*) (*s* : *Symbol*) \Rightarrow
 if (*sym_eq s Epsilon*) then
 map (fun *x* \Rightarrow (*x, snd q*)) (*M4.D (fst q) Epsilon*) ++
 map (fun *y* \Rightarrow (*fst q, y*)) (*M3.D (snd q) Epsilon*) ++
 list_prod (M4.D (fst q) Epsilon) (M3.D (snd q) Epsilon)
 else
 list_prod (M4.D (fst q) s) (M3.D (snd q) s).

Lemma *D_dec* : $\forall (q q' : Qtype) (s : Symbol),$
 $\{In q' (D q s)\} + \{\sim(In q' (D q s))\}$.

Fixpoint *beq_q* (*q q' : Qtype*) { *struct q'* } : *bool* :=
 match *q, q'* with
 | (*q4, q3*), (*q4', q3'*) \Rightarrow *andb (M4.beq_q q4 q4') (M3.beq_q q3 q3')*
 end.

Lemma *beq_q_eq* : $\forall q q' : Qtype, q = q' \leftrightarrow true = beq_q q q'$.

Lemma *beq_q_refl* : $\forall q : Qtype, true = beq_q q q$.

End *M5'*.

Module *M5* := *NFA M5'*.

Lemma *m5_q_cross* : $\forall (q4 : M4.Qtype) (q3 : M3.Qtype),$
 $In q4 M4.Q \rightarrow In q3 M3.Q \rightarrow In (q4, q3) M5.Q$.

Lemma *m5_q_cross2* : $\forall (q4 : M4.Qtype) (q3 : M3.Qtype),$
 $In (q4, q3) M5.Q \rightarrow In q4 M4.Q \wedge In q3 M3.Q$.

Lemma *m5_eq_m4_eq* : $\forall (a b : M4.Qtype \times M3.Qtype),$
 $a = b \rightarrow fst a = fst b$.

Lemma *m5_eq_m3_eq* : $\forall (a b : M4.Qtype \times M3.Qtype),$
 $a = b \rightarrow snd a = snd b$.

Lemma *m5_d_cross2* : $\forall (q4\ q4' : M4.Qtype) (q3\ q3' : M3.Qtype) (c : Char),$
 $In\ (q4',\ q3')\ (M5.D\ (q4,\ q3)\ (Character\ c)) \rightarrow$
 $In\ q4'\ (M4.D\ q4\ (Character\ c)) \wedge In\ q3'\ (M3.D\ q3\ (Character\ c)).$

Lemma *m5_d_cross2_epsilon* :
 $\forall (q4\ q4' : M4.Qtype) (q3\ q3' : M3.Qtype),$
 $In\ (q4',\ q3')\ (M5.D\ (q4,\ q3)\ Epsilon) \rightarrow$
 $(In\ q4'\ (M4.D\ q4\ Epsilon) \wedge In\ q3'\ (M3.D\ q3\ Epsilon))$
 $\vee (In\ q4'\ (M4.D\ q4\ Epsilon) \wedge q3 = q3')$
 $\vee (In\ q3'\ (M3.D\ q3\ Epsilon) \wedge q4 = q4').$

Lemma *m5_d_cross_epsilon_m4* : $\forall (q4\ q4' : M4.Qtype) (q3\ q3' : M3.Qtype),$
 $In\ q4'\ (M4.D\ q4\ Epsilon) \rightarrow q3 = q3' \rightarrow$
 $In\ (q4',\ q3')\ (M5.D\ (q4,\ q3)\ Epsilon).$

Lemma *m5_d_cross_epsilon_m3* : $\forall (q4\ q4' : M4.Qtype) (q3\ q3' : M3.Qtype),$
 $In\ q3'\ (M3.D\ q3\ Epsilon) \rightarrow q4 = q4' \rightarrow$
 $In\ (q4',\ q3')\ (M5.D\ (q4,\ q3)\ Epsilon).$

Lemma *m5_d_cross*: $\forall (q4\ q4' : M4.Qtype) (q3\ q3' : M3.Qtype) (c : Symbol),$
 $In\ q4'\ (M4.D\ q4\ c) \rightarrow$
 $In\ q3'\ (M3.D\ q3\ c) \rightarrow$
 $In\ (q4',\ q3')\ (M5.D\ (q4,\ q3)\ c).$

Lemma *m5_equiv2* : $\forall (q5\ q5' : M5.Qtype) (s : String),$
 $M5.reachable\ q5\ s\ q5' \rightarrow$
 $M4.reachable\ (fst\ q5)\ s\ (fst\ q5') \wedge$
 $M3.reachable\ (snd\ q5)\ s\ (snd\ q5').$

Axiom *m5_equiv* : $\forall (q5\ q5' : M5.Qtype) (s : String),$
 $M3.reachable\ (snd\ q5)\ s\ (snd\ q5') \rightarrow$
 $M4.reachable\ (fst\ q5)\ s\ (fst\ q5') \rightarrow$
 $M5.reachable\ (fst\ q5,\ snd\ q5)\ s\ (fst\ q5',\ snd\ q5').$

Theorem *m4_concat_works* : $\forall (s\ s' : String),$
 $M1.language\ s \rightarrow$
 $M2.language\ s' \rightarrow$
 $M4.language\ (concat\ s\ s').$

Lemma *m5_intersectworks* : $\forall s : String, M3.language\ s \rightarrow M4.language\ s \rightarrow M5.language\ s.$

Definition *Qlhs* ($q : M5.Qtype$) : Prop := match q with ($q4, -$) $\Rightarrow q4 = InQ1\ M1.f\ end.$

Definition *Qrhs* ($q : M5.Qtype$) : Prop := match q with ($q4, -$) $\Rightarrow q4 = InQ2\ M2.s\ end.$

Definition *SolutionPair* ($q\ q' : M5.Qtype$) : Prop :=
 $In\ q\ M5.Q \wedge In\ q'\ M5.Q \wedge Qlhs\ q \wedge Qrhs\ q' \wedge In\ q'\ (M5.D\ q\ Epsilon).$

Lemma *satisfying_left* :
 $\forall (q : M4.Qtype) (q' : M3.Qtype) (s : String),$
 $Qlhs\ (q,\ q') \rightarrow$
 match q with
 $| InQ1\ q1 \Rightarrow M5.reachable\ M5.s\ s\ (q,\ q') \rightarrow M1.reachable\ M1.s\ s\ q1$
 $| InQ2\ q2 \Rightarrow False$
 end.

Lemma *satisfying_right* :
 $\forall (q : M4.Qtype) (q' : M3.Qtype) (s : String),$
 $Qrhs\ (q,\ q') \rightarrow$
 match q with
 $| InQ1\ q1 \Rightarrow False$
 $| InQ2\ q2 \Rightarrow M5.reachable\ (q,\ q')\ s\ M5.f \rightarrow M2.reachable\ q2\ s\ M2.f$

end.

Theorem *allsolutions* : $\forall (q\ q' : Q5) (s\ s' : String),$
 SolutionPair $q\ q' \rightarrow$
 M5.reachable $M5.s\ s\ q \rightarrow$
 M5.reachable $q'\ s'\ M5.f \rightarrow M5.reachable\ M5.s\ (concat\ s\ s')\ M5.f.$

Bibliography

- [1] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *Principles of Programming Languages*, pages 372–382, 2006.
- [2] Peter Thiemann. Grammar-based analysis of string expressions. In *Workshop on Types in Languages Design and Implementation*, pages 59–70, 2005.
- [3] K. J. Higgins. Cross-site scripting: attackers’ new favorite flaw. Technical report, http://www.darkreading.com/document.asp?doc_id=103774&WT.svl=news1_1, September 2006.
- [4] WhiteHat Security. Whitehat website security statistic report, 8th edition. <http://www.whitehatsec.com>, November 2009.
- [5] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *International Symposium on Static Analysis*, pages 1–18, 2003.
- [6] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *International Conference on the World Wide Web*, pages 432–441, 2005.
- [7] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Programming Language Design and Implementation*, pages 32–41, 2007.
- [8] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Programming Language Design and Implementation*, pages 206–215, June 9–11, 2008.
- [9] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Programming Language Design and Implementation*, pages 213–223, 2005.
- [10] Patrice Godefroid, Michael Levin, and David Molnar. Automated whitebox fuzz testing. In *Network Distributed Security Symposium*, 2008.
- [11] Rupak Majumdar and Ru-Gang Xu. Directed test generation using symbolic grammars. In *Automated Software Engineering*, pages 134–143, 2007.
- [12] Nikolai Tillmann and Jonathan de Halleux. Pex — white box test generation for .net. In *Tests and Proofs*, pages 134–153, 2008.
- [13] Larse Ole Anderson. Program analysis and specialization for the c programming language. ph.d. thesis. Technical report, DIKU Report 94–19, 1994.
- [14] Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. In *Principles of Programming Languages*, pages 327–338, 2007.
- [15] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Principles of Programming Languages*, pages 32–41, 1996.
- [16] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

- [17] Aaron Stump, Clark W. Barrett, and David L. Dill. Cvc: A cooperating validity checker. In *Computer Aided Verification*, pages 500–504, 2002.
- [18] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.
- [19] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *European Systems Conference*, pages 103–122, April 2006.
- [20] A. Aiken and E.L. Wimmers. Solving systems of set constraints. In *Logic in Computer Science*, pages 329–340, jun 1992.
- [21] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing*, pages 502–518, 2003.
- [22] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Design Automation Conference*, pages 530–535, 2001.
- [23] Yichen Xie and Alexander Aiken. Saturn: A SAT-based tool for bug detection. In *Computer Aided Verification*, pages 139–143, 2005.
- [24] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
- [25] George C. Necula. Proof-carrying code. In *Principles of Programming Languages*, pages 106–119, 1997.
- [26] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.
- [27] Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. Fast and precise sanitizer analysis with bek. In *USENIX Security Symposium*, pages 1–15, 2011.
- [28] Margus Veanes, Pieter Hooimeijer, Benjamin Livshits, David Molnar, and Nikolaaj Bjørner. Symbolic finite state transducers: algorithms and applications. In *Principles of Programming Languages*, pages 137–150, 2012.
- [29] Pieter Hooimeijer and Westley Weimer. Modeling bug report quality. In *International Conference on Automated Software Engineering*, pages 73–82, 2007.
- [30] Westley Weimer. Patches as better bug reports. In Stan Jarzabek, Douglas C. Schmidt, and Todd L. Veldhuizen, editors, *Generative Programming and Component Engineering*, pages 181–190. ACM, 2006.
- [31] Michal Kunc. The power of commuting with finite sets of words. *Theory Comput. Syst.*, 40(4):521–551, 2007.
- [32] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *International Conference on Software Engineering*, 2008.
- [33] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004.
- [34] Arto Salomaa, Kai Salomaa, and Sheng Yu. State complexity of combined operations. *Theor. Comput. Sci.*, 383(2-3):140–152, 2007.
- [35] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Symposium on Security and Privacy*, pages 258–263, 2006.

- [36] Michael C. Martin, V. Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 365–383, 2005.
- [37] Y. Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Usenix Security Symposium*, pages 179–192, July 2006.
- [38] British Broadcasting Corporation. UN’s website breached by hackers. In <http://news.bbc.co.uk/2/hi/technology/6943385.stm>, August 2007.
- [39] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: a solver for string constraints. In *International symposium on Software testing and analysis*, pages 105–116, 2009.
- [40] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *IEEE Symposium on Security and Privacy*, pages 513–528, 2010.
- [41] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. Path feasibility analysis for string-manipulating programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 5505 of *LNCS*, pages 307–321. Springer, 2009.
- [42] Margus Veanes, Peli de Halleux, and Nikolai Tillmann. Rex: Symbolic regular expression explorer. In *International Conference on Software Testing, Verification, and Validation*, pages 498–507, 2010.
- [43] Margus Veanes, Nikolaj Bjørner, and Leonardo de Moura. Symbolic automata constraint solving. In *LPAR-17*, volume 6397 of *LNCS/ARCoSS*, pages 640–654. Springer, 2010.
- [44] Michael Sipser. *Introduction to the Theory of Computation*. Second edition. PWS, 1997.
- [45] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.
- [46] J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *TACAS ’95*, volume 1019 of *LNCS*. Springer, 1995.
- [47] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. An automata-based string analysis tool for php. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2010.
- [48] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical J.*, 38(4):985–990, 1989.
- [49] Sheldon B. Akers. Binary decision diagrams. *IEEE Trans. Computers*, 27(6):509–516, 1978.
- [50] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.
- [51] Claude E. Shannon. The synthesis of two-terminal switching circuits. *Bell Systems Technical Journal*, 28:59–98, 1949.
- [52] Pieter Hooimeijer and Westley Weimer. Solving string constraints lazily. In *Automated Software Engineering*, pages 377–386, 2010.
- [53] Nuo Li, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Reggae: Automated test generation for programs using complex regular expressions. In *Automated Software Engineering Short Paper*, November 2009.

- [54] Roland Axelsson, Keijo Heljanko, and Martin Lange. Analyzing context-free grammars using an incremental sat solver. In *International colloquium on Automata, Languages and Programming*, pages 410–422, 2008.
- [55] Lucian Ilie and Sheng Yu. Follow automata. *Inf. Comput.*, 186(1):140–162, 2003.
- [56] Pieter Hooimeijer and Margus Veanes. An evaluation of automata algorithms for string analysis. In *VMCAI*, pages 248–262, 2011.
- [57] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Computer-Aided Verification*, pages 519–531, 2007.
- [58] Randal E. Bryant, Daniel Kroening, Joel Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan Brady. Deciding bit-vector arithmetic with abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 358–372, 2007.
- [59] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Principles of Programming Languages*, pages 58–70, 2002.
- [60] Shuvendu K. Lahiri, Thomas Ball, and Byron Cook. Predicate abstraction via symbolic decision procedures. *Logical Methods in Computer Science*, 3(2), 2007.
- [61] Thomas Ball, Byron Cook, Shuvendu K. Lahiri, and Lintao Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In *Computer Aided Verification*, pages 457–461, 2004.
- [62] Stephen Adams, Thomas Ball, Manuvir Das, Sorin Lerner, Sriram K. Rajamani, Mark Seigle, and Westley Weimer. Speeding up dataflow analysis using flow-insensitive pointer analysis. In Manuel V. Hermenegildo and Germán Puebla, editors, *SAS*, volume 2477 of *Lecture Notes in Computer Science*, pages 230–246. Springer, 2002.
- [63] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Grégoire Sutre, and Westley Weimer. Temporal-safety proofs for systems code. In *Computer Aided Verification*, pages 526–538, 2002.
- [64] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *Computer and Communications Security*, pages 322–335, 2006.
- [65] John Kodumal and Alexander Aiken. Banshee: A scalable constraint-based analysis toolkit. In *Static Analysis Symposium*, pages 218–234, 2005.
- [66] Davide Balzarotti, Marco Cova, Viktoria Felmetzger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE Symposium on Security and Privacy*, pages 387–401, 2008.
- [67] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In *International Symposium on Software testing and analysis*, pages 249–260, 2008.
- [68] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses. In *Automated Software Engineering*, pages 605–609, 2009.
- [69] Fang Yu, Tevfik Bultan, and Oscar H. Ibarra. Relational string verification using multi-track automata. In *CIAA*, pages 290–299, 2010.
- [70] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Patching vulnerabilities with sanitization synthesis. In *International Conference on Software Engineering*, pages 251–260, 2011.

- [71] Fang Yu, Tevfik Bultan, and Oscar H. Ibarra. Symbolic string verification: Combining string analysis and size analysis. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2009.
- [72] K. Lakhotia, P. McMinn, and M. Harman. Automated test data generation for coverage: Haven't we solved this problem yet? In *Testing Academia and Industry Conference*, pages 95–104, September 2009.
- [73] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *International Conference on Software Engineering*, pages 416–426, 2007.
- [74] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In *International Conference on Software Engineering*, pages 1066–1071, 2011.
- [75] Corina S. Pasareanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael R. Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *International Symposium on Software Testing and Analysis*, pages 15–26, 2008.
- [76] Fujitsu Laboratories. Fujitsu develops technology to enhance comprehensive testing of Java programs, 2010.
- [77] Pieter Hooimeijer and Westley Weimer. A decision procedure for subset constraints over regular languages. In *Programming Languages Design and Implementation*, pages 188–198, 2009.
- [78] Gertjan Van Noord and Dale Gerdemann. Finite state transducers with predicates and identities. *Grammars*, 4:2001, 2001.
- [79] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS'08*, LNCS. Springer, 2008.
- [80] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual (C++ In-Depth Series)*. Addison-Wesley Professional, December 2001.
- [81] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA implementation secrets. *International Journal of Foundations of Computer Science*, 13(4):571–586, 2002.
- [82] Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *Programming Language Design and Implementation*, pages 221–231, June 2001.
- [83] Michal Kunc. What do we know about language equations? In *Developments in Language Theory*, pages 23–27, 2007.
- [84] Sebastian Bala. Regular language matching and other decidable cases of the satisfiability problem for constraints between regular open terms. In *STACS*, pages 596–607, 2004.
- [85] Bernard Boigelot and Pierre Wolper. Representing arithmetic constraints with finite automata: An overview. In *ICLP 2002*, pages 1–19.
- [86] Stefan Blom and Simona Orzan. Distributed state space minimization. *J. Software Tools for Technology Transfer*, 7(3):280–291, 2005.
- [87] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2010.