University of California
Santa Barbara

Automata-based Model Counting String Constraint Solver for Vulnerability Analysis

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Abdulbaki Aydın

Committee in charge:

Professor Tevfik Bultan, Chair
Professor Chandra Krintz
Professor Timothy Sherwood

March 2017

The Dissertation of Abdulbaki Aydın is approved.

Professor Chandra Krintz

Professor Timothy Sherwood

Professor Tevfik Bultan, Committee Chair

March 2017

Automata-based Model Counting String Constraint Solver for Vulnerability Analysis

Abdulbaki Aydın

To my lovely wife,

Yasemin,

my parents, and my kids.

Acknowledgements

I would like to thank my advisor, Professor Tevfik Bultan, for his unlimited support I enjoyed during my PhD journey. He is an excellent advisor with amazing technical and soft skills. I was inspired by his passion to push the limits of the state of the art research in computer science. He devoted considerable amount of time discussing ideas with his knowledge, enthusiasm, and motivation. He was always ready to help me no matter the effort implied.

I would also like to thank Professor Chandra Krintz and Professor Timothy Sherwood for serving on my committee and for giving great feedback and vision throughout my PhD journey.

I had the privilege to work with wonderful people at Verfication Lab (VLab). Special thanks to Lucas Bang and Muath Alkhalaf whom I worked closely most of the time. I would like to thank to former VLab member Ivan Bocić and recent VLab members William Eiers, Tegan Brennan, Nicolás Rosner, and Nestan Tsiskaridze for the great research and friendship environment.

My parents, Hüseyin Aydın and Fatma Aydın and my parents-in-law, Necati Şahin and Güler Şahin gave all the blessings and support with love. My sisters and brothers Sümeyra Aydın Özkan, Kübra Aydın, İbrahim Aydın, and Muhammed Aydın, and my sisters-in-law Aybüke Şahin and Ayşegül Şahin Varol were always there to encourage and support me with their unlimited love. I am thankful to God for having such a great family.

I would like to send my love to my son, Eymen Asaf Aydın, who kept me hopeful everyday with his beautiful smiles even though he is too little to understand that.

Finally, I would like to thank my beautiful and lovely wife, Yasemin Şahin Aydın, this was only possible with you by my side.

Curriculum Vitæ
Abdulbaki Aydın

Education

2017	Ph.D. in Computer Science, University of California, Santa Barbara, USA.
2016	M.S. in Computer Science, University of California, Santa Barbara, USA.
2009	B.S. in Computer Engineering, Fatih University, Istanbul, Turkey.
2009	B.S. in Electric-Electronics Engineering, Fatih University, Istanbul, Turkey.

Publications

Abdulbaki Aydin, David Piorkowski, Omer Tripp, Pietro Ferrara, and Marco Pistoia, *Visual Configuration of Mobile Privacy Policies*, in Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, FASE 2017.

Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Pasareanu, and Tevfik Bultan, *String Analysis for Side Channels with Segmented Oracles*, in Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016.

Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan, *Automata-Based Model Counting for String Constraints*, in Proceedings of the 27th International Conference on Computer Aided Verification, CAV 2015.

Lucas Bang, Abdulbaki Aydin, and Tevfik Bultan, *Automatically Computing Path Complexity of Programs*, in Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015.

Abdulbaki Aydin, Muath Alkhalaf, and Tevfik Bultan, *Automated Test Generation from Vulnerability Signatures*, in Proceedings of the 7th IEEE International Conference on Software Testing, Verification and Validation, ICST 2014.

Muath Alkhalaf, Abdulbaki Aydin, and Tevfik Bultan, *Semantic Differential Repair for Input Validation and Sanitization*, in Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014.

Abstract

Automata-based Model Counting String Constraint Solver for Vulnerability Analysis

by

Abdulbaki Aydın

Most common vulnerabilities in modern software applications are due to errors in string manipulation code. String constraint solvers are essential components of program analysis techniques for detecting and repairing vulnerabilities that are due to string manipulation errors. In this dissertation, we present an automata-based string constraint solver for vulnerability analysis of string manipulating programs.

Given a string constraint, we generate an automaton that accepts all solutions that satisfy the constraint. Our string constraint solver can also map linear arithmetic constraints to automata in order to handle constraints on string lengths. By integrating our string constraint solver to a symbolic execution tool, we can check for string manipulation errors in programs. Recently, quantitative and probabilistic program analyses techniques have been proposed which require counting the number of solutions to string constraints. We extend our string constraint solver with model counting capability based on the observation that, using an automata-based constraint representation, model counting reduces to path counting, which can be solved precisely. Our approach is parameterized in the sense that, we do not assume a finite domain size during automata construction, resulting in a potentially infinite set of solutions, and our model counting approach works for arbitrarily large bounds.

We have implemented our approach in a tool called ABC (Automata-Based model Counter) using a constraint language that is compatible with the SMTLIB language specification used by satisfiability-modulo-theories solvers. This SMTLIB interface fa-

cilitates integration of our constraint solver with existing symbolic execution tools. We demonstrate the effectiveness of ABC on a large set of string constraints extracted from real-world web applications.

We also present automata-based testing techniques for string manipulating programs. A vulnerability signature is a characterization of all user inputs that can be used to exploit a vulnerability. Automata-based static string analysis techniques allow automated computation of vulnerability signatures represented as automata. Given a vulnerability signature represented as an automaton, we present algorithms for test case generation based on state, transition, and path coverage. These automatically generated test cases can be used to test applications that are not analyzable statically, and to discover attack strings that demonstrate how the vulnerabilities can be exploited. We experimentally compare different coverage criteria and demonstrate the effectiveness of our test generation approach.

Contents

Chapter 1

Introduction

Support for string manipulation in programming languages has been increasing due to the common usage of strings in modern software applications. For example, many modern software applications are web applications, and most of what web applications do is string processing. Common uses of string manipulation in modern software development are as follows:

- *Input validation and sanitization:* Most modern software applications are web-based and the user inputs to web applications typically come in the string form. The input string entered by the user is parsed by the web application and can be used as the input parameter for the action that is executed in response to the user's request. Web applications are globally accessible and any user can interact with them from all over the world. A user can input a string in an undesired format or a malicious user can input a string that contains hidden commands. Application developers cannot trust that user input is valid and safe; they need to validate and/or sanitize user input to keep application and its data in a consistent state and to avoid vulnerabilities. Input validation rejects user input if it is not in desired format or if it contains harmful string patterns. Input sanitization modifies

user input string to transform it into a safe and valid input parameter to the web application. Both input validation and sanitization involves extensive string manipulation since user inputs are typically in string form.

- *Database query generation:* Many modern software applications have a back-end database component to store data. Many user interactions with the application can trigger database queries, which are constructed at runtime using string manipulation.

- *Formatted data generation:* Modern software applications commonly use well-known data formats to store, exchange, or describe data. XML and JSON are two of the widely used data formats. HTML is the most common format worldwide to describe web documents and to display user input forms in web applications. Creation of such data formats involves extensive string manipulation.

- *Dynamic code generation:* Many modern software applications are highly dynamic where application code is generated on the fly based on user input. In case of web applications, dynamically generated code can correspond to client side code (e.g., JavaScript) or in some cases it can correspond to server side code. In both cases, dynamic code generation requires string manipulation.

In all use cases listed above, errors in string manipulation code can have disastrous effects. Analyzing string manipulation code is extremely important in order to avoid failures in modern software. String analysis aims to automatically analyze string manipulation code to check its correctness, based on the developer's expectations. It is a static program analysis technique that determines the values that a string expression can take during program execution. String analysis can be used to solve many problems in modern software systems that relate to string manipulation. It can be applied, e.g., to identify

security vulnerabilities by checking if a security sensitive function receives an input that contains an exploit [1, 2, 3, 4], to identify set of user inputs that reaches to a sensitive function with certain values [5], to generate test cases from set of string values of a string expression [6], to identify data format generation errors [7], to identify the set of queries that are sent to back-end database [8, 2], to identify set of dynamically generated code (e.g., client-side code) [9, 10, 11], and to generate patches for string manipulation code (e.g., patching input validation and sanitization functions) [12, 13].

Like many other program-analysis problems, it is not possible to solve the string analysis problem precisely (i.e., it is not possible to precisely determine the set of string values that can reach a program point). However, one can compute over- or under-approximations of possible string values. If the approximations are precise enough, they can enable us to demonstrate existence or absence of bugs in string manipulating code.

String analysis has been an active research area in the last decade, resulting in a wide variety of string-analysis techniques such as, grammar-based string analysis [7, 14], automata-based symbolic string analysis [15, 16, 17, 18, 19], string constraint solving [20, 21, 22, 23, 24, 25], string abstractions [26, 27], relational string analysis [28], vulnerability detection using string analysis [1, 2, 29], differential string analysis [30, 13], and automated repair using string analysis [12, 13]. Two main string analysis approaches are 1) symbolic verification techniques based on fixpoint computations, and 2) symbolic execution techniques based on constraint solving.

There are two recent research directions that aim to extend symbolic execution beyond assertion checking. One of them is quantitative information flow, where the goal is to determine how much secret information is leaked from a given program [31, 32, 33, 34], and another one is probabilistic symbolic execution where the goal is to compute probabilities of program execution paths in order to establish reliability of the given program [35, 36]. Interestingly, both of these approaches require the same basic extension to con-

straint solving: They require a model-counting constraint solver that not only determines if a constraint is satisfiable, but also computes the number of satisfying instances. This is known as the *model counting* problem.

In this dissertation, we show that automata-based techniques are effective in handling mixed string and integer constraint solving and model counting problems that arise in symbolic analysis of string manipulating programs.

The rest of this dissertation presents the following contributions:

- In Chapter 2 we provide background information on the state of the art techniques and give examples of model counting applications.
- In Chapter 3 we present a core string constraint language and demonstrate that this constraint language can capture string constraints from multiple programming languages.
- In Chapter 4 we discuss the techniques we developed for mapping string constraints to automata.
- In Chapter 5 we discus the techniques we developed for handling relational and mixed constraints using multi-track automata.
- In Chapter 7 we discuss automata-based model counting.
- In Chapter 8 we discuss the ABC tool and provide experiments demonstrating the effectiveness of automata-based string constraint solving and model-counting.
- In Chapter 9 we discuss automata-based test case generation for string manipulating programs.

Finally, in Chapter 10, we conclude the work.

Chapter 2

Background

Analysis of string manipulating programs has been studied extensively in recent years. In this chapter, we first provide illustrative examples of the two commonly used string analysis techniques: *symbolic verification* and *symbolic execution*. Then, we discuss current string constraint solving techniques. Lastly we provide examples of model counting applications to motivate our work.

2.1 Symbolic Verification

Symbolic verification is a general program analysis technique to verify correctness of programs. In symbolic verification, programs are analyzed symbolically using an abstraction that depends on the verification problem. String manipulation errors that relate to input validation or sanitization can be detected using symbolic verification techniques.

Input validation or sanitization is crucial for web applications. If input validation or sanitization is not used, inputs that violate the expected format can easily cause an application to crash since the user input becomes the input parameter of the action that is executed based on the user request. Moreover, during action execution, user input can

```php
<?php
 function escdata ($data) {
     global $dbc;
     if (ini_get('magic_quotes_gpc')) {
         $data = stripslashes($data);
     }
     return mysql_real_escape_string(trim ($data), $dbc);
 }

 function xss_clean ($var) {
     $var = preg_replace('/[Jj][Aa][Vv][Aa][Ss][Cc][Rr][Ii][Pp][Tt]/', '
    java script', $var );
     return $var;
 }

 $titlee = escdata(xss_clean($_POST['title']) );
 echo "<p>Title:<br />" . $titlee . "</p>";
?>
```

Figure 2.1: A PHP sanitization example.

be passed as a parameter to security sensitive operations such as sending a query to the back-end database. In order to ensure the security of the application, the user inputs that flow into security sensitive functions must be correctly validated and sanitized. Due to global accessibility of web applications, malicious users all around the world can exploit a vulnerable application, so any existing vulnerability in a web application is likely to be exploited by some malicious user somewhere. Given the significance of this security threat, one would expect web application developers to be extremely careful in writing input validation and sanitization functions. Unfortunately, web applications are notorious for security vulnerabilities such as SQL injection and cross-site scripting (XSS) that are due to improper input validation and sanitization.

As an illustrative example, Figure 2.1 represents a simplified version of a real world input sanitization example from a database driven knowledge base management applica-

tion called Andy's PHP Knowledge Base [1] to avoid XSS attacks. User input is read at line 15 from global parameter `$_POST['title']` and first passed onto `xss_clean` function. Next, the result of `xss_clean` function call is passed onto `escdata` function for further sanitization. Finally, the resulting string is used to generate an HTML code as a response to the user request. Unfortunately, sanitization done on the input string is not sufficient enough to avoid XSS attacks. It allows malicious users to inject arbitrary javascript code (via user input), which leads to `<script` tag to flow into a sink (i.e., a security sensitive function).

Automata-based static string analysis, which is a type of symbolic verification technique, has been used to check for vulnerabilities such as XSS [29]. Automata-based analysis encodes the set of string values that string variables can take at any point during program execution as automata. This information can be used to verify that string values are sanitized properly and to detect programming errors and security vulnerabilities.

Automata-based static string analysis works on top of a data dependency graph of the program that is being analyzed. A data dependency graph is a directed graph representing dependencies between program points as a flow from program entry points to security sensitive functions (i.e., sinks). Figure 2.2 shows the data dependency graph for the PHP code presented in Figure 2.1. In the example, `echo` function is treated as a security sensitive function (i.e., a sink) and the data dependency graph represents all program points that affect possible string values that appear as a parameter to the selected sensitive function. The data nodes are represented with light blue rectangles and the operation nodes are represented with ellipses in the data dependency graph. Each node in the graph has a subscript that represents topological sort order of the node. Automata-based symbolic string analysis creates an automaton for each node by traversing the data

[1] `http://aphpkb.org/`

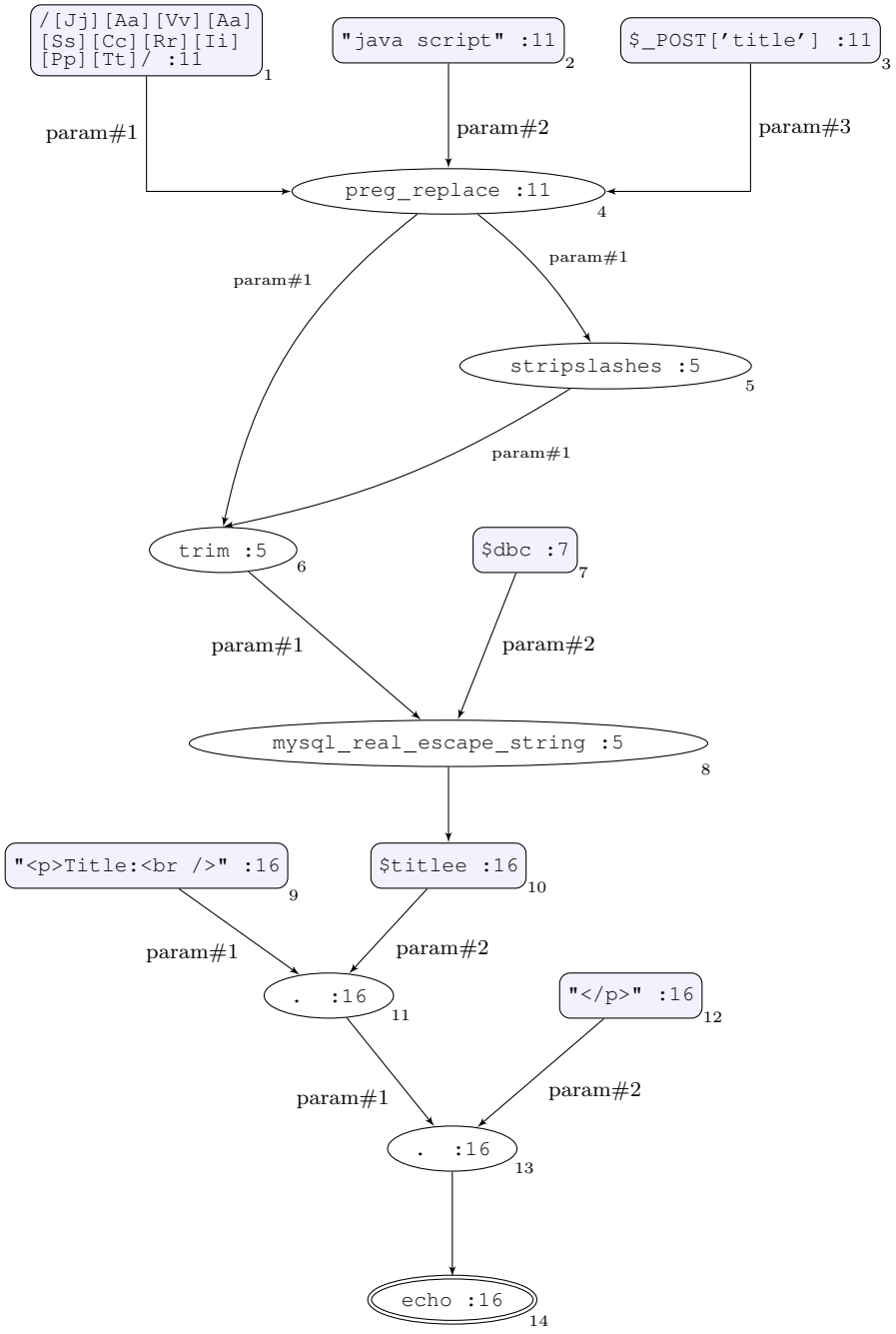

Figure 2.2: Data dependency graph of the example code in Figure 2.1.

dependency graph in topological order. First node in the example corresponds to the regular expression pattern `/[Jj][Aa][Vv][Aa][Ss][Cc][Rr][Ii][Pp][Tt]/` and an automaton that accepts the set of strings defined by the regular expression is created for this node. Similarly, an automaton that accepts the string `"java script"` is created for the second node. Third node corresponds to a public user input. A user can enter any string value as an input to a web application. In other words, public inputs are initially unrestricted, thus an automaton that accepts any string is generated for the third node. Fourth node is an operation node which corresponds to PHP built-in function `preg_replace`. Automata-based string analysis uses symbolic versions of built-in string manipulating functions. These symbolic functions accept automata as inputs (characterizing possible input strings) and return an automaton as the result (characterizing all possible output strings based on the given input strings). Symbolic version of `preg_replace` function accepts the first three automata generated for the first three nodes as parameters. It computes an automaton for the fourth node as a result of the symbolic computation. There may be cases where a node receives data flows from different sources for the same target. `trim` node, the sixth node in this example, gets two different flows; one from the result of `preg_replace` node and one from the result of `stripslashes` node for its only parameter. In such cases, union of all the automata that are computed for source nodes is passed as a parameter to target node. Automata computation for each node continues in the same way described above in topological order until an automaton is created for each node in the data dependency graph. Note that seventh node in the graph does not have any effect on string values. Such nodes are discarded during automata computation. For the sink node, `echo` call at line 16, an automaton that represents all the values that flow into the sink function is generated. To do vulnerability check one has to specify harmful strings that can exploit a security vulnerability at the sink location. Harmful input strings can be specified with a regular

expression which is also known as an attack pattern. To simplify our discussion with the running example, we can assume an attack pattern as a set of all strings that contain an open script tag (e.g., `/.*<.*/`). An automaton that accepts all strings defined by the attack pattern is generated. Vulnerability check is done by checking the emptiness of the intersection of the attack pattern automaton and the sink node automaton. In the example we have, the intersection automaton is not empty. In that case, symbolic string analysis conclude that the application is vulnerable to certain attacks defined by the attack pattern.

As shown in the example, automata-based string analysis techniques can be used to check for security vulnerabilities in web applications. Automata-based symbolic string analysis can handle more complex programs such as the ones with loop constructs. A loop in a program may correspond to a cycle in the data dependency graph. In such cases, strongly connected components are identified in the data dependency graph. Automata-based symbolic string analysis provides a widening operator [37, 29] defined on automata to approximate fixpoint computations to handle loops. A strongly connected component can be treated as a single node in the data dependency graph by computing an automaton for it with the help of the automata widening operator. With that, cyclic data dependency graphs can be handled, i.e., programs with loop constructs can be analyzed. Automata-based symbolic string analysis can also handle unbounded strings, because set of string values are represented by regular languages. With the advantages of automata-based symbolic string representation, the vulnerability analysis described here can be extended to other types of vulnerabilities using similar techniques.

2.2 Symbolic Execution

Symbolic execution is a program analysis technique to determine what input values cause each path of a program to execute [38]. Symbolic execution assumes symbolic values for the program inputs rather than using concrete values as normal execution of the program would. Expressions encountered during symbolic execution are expressed as functions of the symbolic variables. At any point during symbolic execution, program state is described with the value of the program counter and with a symbolic expressions known as the path condition (PC). A PC is a constraint on input values that must be satisfied in order for a program to reach the location that PC corresponds. The set of all possible executions of a program is represented by a symbolic execution tree.

Figure 2.3 shows a JAVA string manipulation example originally presented as a command injection example in an earlier work [39] and used as a string constraint solving example in Symbolic Path Finder[2] (SPF). It is part of the WU-FTPD implementation of the file transfer protocol. It is originally written in C programming language and the example is converted from the original version. If the input command contains substring `"%n"` an exception is thrown at line 22. In the original implementation, this situation would allow user to alter program stack and gain privileged access to server running the program. This example demonstrates one application of symbolic execution, it checks feasibility of program execution paths that may lead to a vulnerability.

Figure 2.4 represents the symbolic execution tree of `site_exec` function in the example. Symbolic execution tree is built on the fly based on a traversal strategy (e.g., depth-first, breadth-first). Symbolic execution tree of our running example is created by exploring program paths with depth-first exploration strategy. Blue rectangles represent updates on string expressions and diamonds represent branch points encountered during

[2] `http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc`

```
1  public void site_exec(String cmd) {
2    String p = "home/ftp/bin";
3    int j, sp = cmd.indexOf(' ');
4
5    if (sp == -1) {
6      j = cmd.lastIndexOf('/');
7    } else {
8      j = cmd.lastIndexOf('/', sp);
9    }
10
11   String r = cmd.substring(j);
12   int l = r.length() + p.length();
13
14   if (l > 32) {
15     return;
16   }
17
18   String buf = p + r;
19   boolean t = buf.contains("%n");
20
21   if (t == true) {
22     throw new Exception("THREAT");
23   }
24
25   execute(buf);
26   return;
27 }
```

Figure 2.3: A Java string manipulation example.

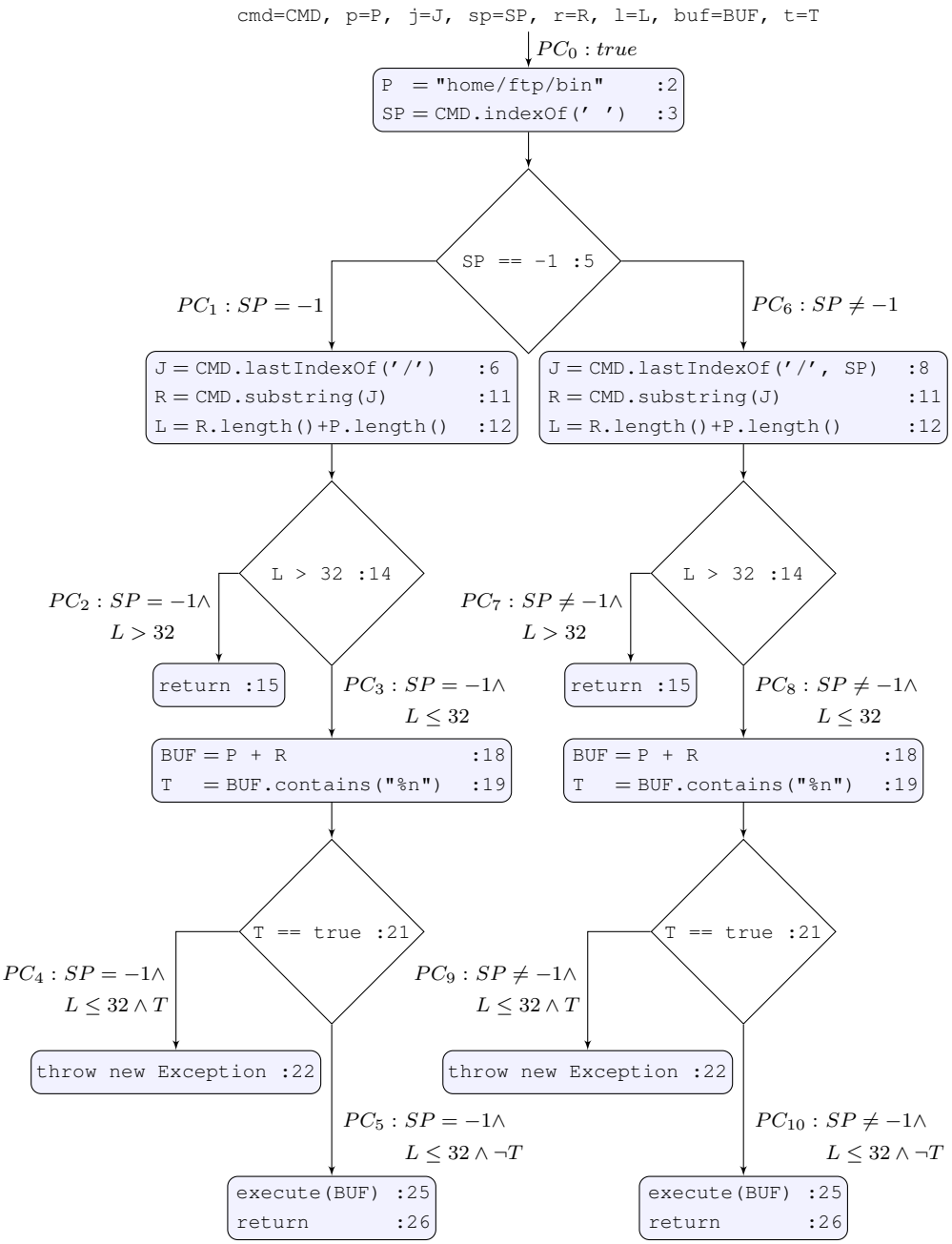

Figure 2.4: Symbolic execution tree of the example code in Figure 2.3.

symbolic execution. Initially, all program variables are replaced with symbolic variables, `cmd=CMD, p=P, j=J, sp=SP, r=R, l=L, buf=BUF, t=T` and initial PC is set to true, $PC_0 = true$. At line 3 symbolic variable `P` is assigned to concrete string with value `"home/ftp/bin"`. At line 4 symbolic variable `SP` is assigned to a function of symbolic variable `CMD`. Line 6 corresponds to a branch point where it checks the condition on symbolic variable `SP`. In order to continue with the symbolic execution, PC is updated with constraints on the symbolic variables for alternative paths, i.e., $PC_1 : SP = -1$ and later on $PC_6 : SP \neq -1$ are generated. Satisfiability of PC_1 is checked using string constraint solvers; if it is satisfiable, symbolic execution continues to explore deeper. Otherwise, if a path condition is unsatisfiable, symbolic execution does not continue for that path and it backtracks and checks satisfiability of alternative PCs.

In the example, the symbolic execution tree shows six different feasible paths represented by path constraints PC_2, PC_4, PC_5, PC_7, PC_9, and PC_{10}. Among those PC_4 and PC_9 characterizes concrete program executions that can exploit the vulnerability with the following conditions:

$$PC_4 : SP = -1 \wedge L \leq 32 \wedge T$$

$$PC_9 : SP \neq -1 \wedge L \leq 32 \wedge T$$

where each PC can be expanded by writing them as function of symbolic variable CMD:

$$PC_4 : CMD.\text{indexOf}(' ') = -1 \wedge CMD.\text{substring}(CMD.\text{lastIndexOf}('/')).\text{length}() + \text{“home/ftp/bin”}.\text{length}() \leq 32 \wedge (\text{“home/ftp/bin”} + CMD.\text{substring}(CMD.\text{lastIndexOf}('/'))).\text{contains}(\text{“\%}n\text{”})$$

$$PC_9 : CMD.\text{indexOf}(' ') \neq -1 \wedge CMD.\text{substring}(CMD.\text{lastIndexOf}('/', CMD.\text{indexOf}(' '))).\text{length}() + \text{“home/ftp/bin”}.\text{length}() \leq 32 \wedge (\text{“home/ftp/bin”} + CMD.\text{substring}(CMD.\text{lastIndexOf}('/', CMD.\text{indexOf}(' ')))).\text{contains}(\text{“\%}n\text{”})$$

Expanded versions of path constraints PC_4 and PC_9 shows that string constraints can be mixed with integer constraints. There are also complex string functions such as lastIndexOf and substring where the result of the former can be a parameter to the latter as in the above PCs. Type of the constraints that we get from the example shows that string constraint solving is essential for symbolic execution of string manipulating programs.

2.3 String Constraint Solving

Symbolic execution can be applied to string manipulating programs. However, symbolic execution of string manipulating programs is difficult since solving string constraints is a challenging problem. String constraint solving is challenging due to two main reasons: 1) String constraints are usually mixed with integer constrains which requires solving integer constraints together with string constraints. 2) With the increasing usage of strings in modern software development, programming languages provide increasingly complex string operations that need to be handled by string constraint solvers.

Several string constraint solvers (satisfiability checkers) have been proposed in re-

cent years. There are two main approaches: 1) bit-vector based bounded checking [26, 14, 20, 22], and 2) satisfiability modulo theories (SMT) based constraint solving [21, 40, 41, 24, 23]. Bit-vector based solvers support core string operations such as equality, membership, concatenation, and string length equations. Additional complex operations can be encoded using core string operations to some extent. SMT based constraint solvers support core string operations, and, can also support some complex string operations. Both approaches are good at solving string constraints with length equations, i.e., bounding the length of the strings allows satisfiability checkers to solve length equations efficiently. Bit-vector based solvers have limited support for mixed string constraints. Compared to the bit-vector based approaches, SMT solvers support several different theories and they are more expressive in terms of mixed constraints.

Theories supported by SMT solvers are standardized by SMTLIB[3] community. There are no standards defined for string theory yet. String constraint solving is still an active research area that is evolving.

2.4 Model Counting

Model counting is an extension to constraint solving where instead of just answering to the question "*Does there exist a model that satisfies a given constraint?*" it also tries to answer "*How many models are there that satisfies a given constraint?*" We provide an example below to demonstrate a use case for model counting.

How strong is my password policy? Consider the example in Figure 2.5 which is a C string manipulation example that is originally presented as a use case of model counting [42]. On UNIX, users use the PASSWD utility to change their passwords. The example

[3] `http://smtlib.cs.uiowa.edu/`

```
1  static int string_checker_helper (const char* p1, const char* p2) {
2    if (strcasestr(p2, p1) != NULL || strcasestr(p1, p2) {
3      return 1;
4    }
5    return 0;
6  }
7
8  static int string_checker (const char* p1, const char* p2) {
9    ...
10   int ret = string_checker_helper(p1, p2); ...
11   char* p = reverse_of(p1); ...
12   ret |= string_checker_helper(p, p2); ...
13   return ret;
14 }
15
16 static const char* obscure_msg(const char* old_p, const char* new_p, const
       struct passed* pw) {
17   ...
18   if (old_p && old_p[0] != '\0'){
19     if (string_checker(new_p, old_p)) {
20       return "similar to old password";
21     }
22   }
23   ...
24   return NULL;
25 }
```

Figure 2.5: A Java string manipulation example.

is a simplified version of a C code called OBSCURE which is used by PASSWD utility to check the password strength. At line 1, `string_checkher_helper` functions checks if any of the parameters is a substring of the other one (`strcasestr` function works as a case insensitive substring check). At line 8, `string_checker` function calls `string_checkher_helper` function twice; first with the original parameters and then by reversing the input parameter p_1. If strength check fails, `obscure_msg` function warns user for the similarity to the old password.

Suppose an attacker learns old password and the constraints imposed on new password by OBSCURE utility. The model counting question is, how many possible new password values are there for the attacker to try?

Let us assume old password is "abc-16" and attacker is trying to estimate the number of all possible new passwords. The obscure function checks if one does not contain the other or its reverse in a case insensitive manner. The example code updates the password only if the new password is not too similar to the old one. A symbolic execution tool can identify PCs that result in password update, i.e., the relation between new password and old password can be expressed in terms of a PC. For example, the following is a path constraint that leads to password update:

$$\texttt{strcasestr(NEW\_P, "abc-16")} = \texttt{NULL} \wedge$$
$$\texttt{strcasestr("abc-16", NEW\_P)} = \texttt{NULL} \wedge$$
$$\texttt{strcasestr(NEW\_P, "61-cba")} = \texttt{NULL} \wedge$$
$$\texttt{strcasestr("61-cba", NEW\_P)} = \texttt{NULL}$$

A string model counter can count number of solutions to symbolic variable `NEW_P` that satisfies the given PC. This information can be used for inferring the value of the new password if an attacker knows the value of the old password. Using model counting, one

can assess the likelihood of an attacker guessing the new password, hence, can evaluate the strength of the password policy.

Chapter 3

Constraint Language

In this chapter we present a core string constraint language that can be used to represent string and numeric constraints that result from string analysis. We designed this language to be rich enough to capture constraints from multiple languages. We demonstrate mapping of string expressions from multiple languages (Java and PHP) to our core string constraint language.

3.1 Language Semantics

We define the set of string and linear integer arithmetic constraints using the abstract grammar presented in Figure 3.1. In our constraint language, φ denotes a formula, β denotes an integer term, γ denotes a string term, ρ denotes a constant regular expression, n denotes an integer constant, $\top$ and $\bot$ denote constants true and false, and v_i, v_s denote integer and string variables, respectively. An *atomic constraint* refers to a formula without any boolean connective. $\varphi_{\mathbb{Z}}$ and $\varphi_{\mathbb{S}}$ denote atomic integer and string constraints, respectively. Notice that an integer term produced from the production rule β may contain string terms γ and vice versa; a constraint produced in this way is called a *mixed*

$$
\begin{aligned}
\varphi &\longrightarrow \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \mid \varphi_{\mathbb{Z}} \mid \varphi_{\mathbb{S}} \mid \top \mid \bot \\
\varphi_{\mathbb{Z}} &\longrightarrow \beta = \beta \mid \beta < \beta \mid \beta > \beta \\
\varphi_{\mathbb{S}} &\longrightarrow \gamma = \gamma \mid \gamma < \gamma \mid \gamma > \gamma \mid \text{match}(\gamma, \rho) \mid \text{contains}(\gamma, \gamma) \mid \text{begins}(\gamma, \gamma) \mid \text{ends}(\gamma, \gamma) \\
\beta &\longrightarrow v_i \mid n \mid \beta{+}\beta \mid \beta{-}\beta \mid \beta{\times}n \mid \text{length}(\gamma) \mid \text{toint}(\gamma) \mid \text{indexof}(\gamma, \gamma) \mid \text{lastindexof}(\gamma, \gamma) \\
\gamma &\longrightarrow v_s \mid \rho \mid \gamma \cdot \gamma \mid \text{reverse}(\gamma) \mid \text{tostring}(\beta) \mid \text{charat}(\gamma, \beta) \mid \text{toupper}(\gamma) \mid \text{tolower}(\gamma) \\
&\quad \mid \text{substring}(\gamma, \beta, \beta) \mid \text{replacefirst}(\gamma, \gamma, \gamma) \mid \text{replacelast}(\gamma, \gamma, \gamma) \mid \text{replaceall}(\gamma, \gamma, \gamma) \\
\rho &\longrightarrow \varepsilon \mid s \mid \rho \cdot \rho \mid \rho \mid \rho \mid \rho^*
\end{aligned}
$$

Figure 3.1: Constraint language grammar.

constraint.

We define Σ to denote the set of all characters (i.e., the alphabet), $s \in \Sigma^*$ denote a string value and ε denotes empty string. A character is a string that has length one. The string operations "$\cdot$", "$\mid$", "$*$" correspond to regular expression operations concatenation, alternation, and Kleene closure, respectively. Other regular expression operators (e.g., closure, repetition) can be defined using the existing ones. "$<$" and "$>$" operations on string terms correspond to lexicographical string comparisons.

A *predicate operation* refers to a string or an integer operation that returns a boolean value. "=", "$<$", "match", and "contains" are examples of predicate operations. A *term operation* refers to a string or an integer operation that returns a string or an integer value. "+", "$\cdot$", "length", and "substring" are examples of term operations.

The set of variables present in φ is given by $\mathcal{V}(\varphi)$. A *model* for φ is an assignment of all variables in $\mathcal{V}(\varphi)$ such that φ evaluates to *true*. The truth set of a formula φ, denoted $[\![\varphi]\!]$, is the set of all models of φ. Our eventual goal is to determine the set $[\![\varphi]\!]$ and its size.

Let s, t, v be string values, i, j, n be integer values, and p be a regular expression where

$\mathcal{L}(p)$ denote set of strings (i.e., the language) defined by the regular expression p. Let $|s|$ denote the length of string s; i.e., length$(s) = |s|$. We define the semantics of string related operations as follows:

- match(v, p) returns true if v matches the regular expression expression p; otherwise, returns false. Let $\mathcal{L}(p)$ denote set of strings (i.e., the language) defined by the regular expression p, then we define the semantics of match as follows:

$$\text{match}(v, p) \Leftrightarrow v \in \mathcal{L}(p)$$

- contains(v, t) returns true if the string t is a substring of the string v; otherwise, returns false.

$$\text{contains}(v, t) \Leftrightarrow \exists s_1, s_2 \in \Sigma^* \colon v = s_1 t s_2$$

- begins(v, t) returns true if the string v begins with the string t; otherwise, returns false.

$$\text{begins}(v, t) \Leftrightarrow \exists s \in \Sigma^* \colon v = ts$$

- ends(v, t) returns true if the string v ends with the string t; otherwise, returns false.

$$\text{ends}(v, t) \Leftrightarrow \exists s \in \Sigma^* \colon v = st$$

- length(v) returns the length of the string v.

$$(\text{length}(v) = 0 \Leftrightarrow v = \varepsilon) \wedge$$
$$(\text{length}(v) = n \Leftrightarrow \exists s_0, s_1, \ldots, s_{n-1} \in \Sigma \colon v = s_0 s_1 \ldots s_{n-1})$$

- toint(v) returns the corresponding integer value of the string v. Let $f = \{(\text{`0'}, 0),$

$(\text{`1'}, 1), (\text{`2'}, 2), (\text{`3'}, 3), (\text{`4'}, 4), (\text{`5'}, 5), (\text{`6'}, 6), (\text{`7'}, 7), (\text{`8'}, 8), (\text{`9'}, 9)\}$ be a function that maps each digit character to the corresponding integer value, then we define the semantics of toint as follows:

$$\begin{aligned}\text{toint}(v) = n \Leftrightarrow \exists s_0, s_1, \ldots, s_i \in \{&\text{`0'}, \text{`1'}, \text{`2'}, \text{`3'}, \text{`4'}, \text{`5'}, \text{`6'}, \text{`7'}, \text{`8'}, \text{`9'}\} : \\ &(v = s_0 s_1 \ldots s_i \wedge n = \sum_{j=0}^{i} f(s_j) \times 10^{i-j}) \vee \\ &(v = \text{`-'} s_0 s_1 \ldots s_i \wedge n = -\sum_{j=0}^{i} f(s_j) \times 10^{i-j})\end{aligned}$$

- indexof(v, t) returns the index within the string v of the first occurrence of the string t. If t is not a substring of v, it returns -1.

$$\begin{aligned}&(\text{indexof}(v,t) = -1 \quad \Leftrightarrow \neg\text{contains}(v,t) \wedge \\ &(\text{indexof}(v,t) = \ \ 0 \quad \Leftrightarrow t = \varepsilon) \wedge \\ &(\text{indexof}(v,t) = \ \ n \quad \Leftrightarrow \exists s_1, s_2 \in \Sigma^* \colon v = s_1 t s_2 \wedge \neg\text{contains}(s_1, t) \wedge n = |s_1|)\end{aligned}$$

- lastindexof(v, t) returns the index within the string v of the last occurrence of the string t, If t is not a substring of v, it returns -1.

$$\begin{aligned}&(\text{lastindexof}(v,t) = -1 \quad \Leftrightarrow \neg\text{contains}(v,t) \wedge \\ &(\text{lastindexof}(v,t) = \ |v| \quad \Leftrightarrow t = \varepsilon) \wedge \\ &(\text{lastindexof}(v,t) = \ \ n \quad \Leftrightarrow \exists s_1, s_2 \in \Sigma^* \colon v = s_1 t s_2 \wedge \neg\text{contains}(s_2, t) \wedge n = |s_1|)\end{aligned}$$

- reverse(v) returns the reverse of the string v.

$$\text{reverse}(v) = t \Leftrightarrow \exists s_0, s_1, \ldots, s_i \in \Sigma \colon v = s_0 s_1 \ldots s_i \wedge t = s_i \ldots s_1 s_0$$

- tostring(n) returns the corresponding string value of the integer n. Let $f = \{(0, \text{`0'}), (1, \text{`1'}), (2, \text{`2'}), (3, \text{`3'}), (4, \text{`4'}), (5, \text{`5'}), (6, \text{`6'}), (7, \text{`7'}), (8, \text{`8'}), (9, \text{`9'})\}$ be a function that maps each single digit natural number to the corresponding string representation, then we define the semantics of tostring as follows:

$$\begin{aligned}\text{tostring}(n) = v \Leftrightarrow \exists n_0, n_1, \ldots, n_i \in \{0,1,2,3,4,5,6,7,8,9\} : \\ (n = \sum_{j=0}^{i} n_j \times 10^{i-j} \wedge v = f(n_0)f(n_1)\ldots f(n_i)) \vee \\ (n = -\sum_{j=0}^{i} n_j \times 10^{i-j} \wedge v = f(n_0)f(n_1)\ldots f(n_i))\end{aligned}$$

- charat(v, i) returns the character that appears at the index i of the string v. The semantics of charat is defined as follows:

$$\text{charat}(v,i) = t \Leftrightarrow \exists s_0, s_1, \ldots, s_n \in \Sigma\text{: } v = s_0 s_1 \ldots s_n \wedge 0 \leq i \leq n \wedge t = s_i$$

- toupper(v) transforms all characters of the string v into upper-case characters and returns the result. Let $f = \{(\text{`}a\text{'}, \text{`}A\text{'}), (\text{`}b\text{'}, \text{`}B\text{'}), (\text{`}c\text{'}, \text{`}C\text{'}), \ldots, (\text{`}z\text{'}, \text{`}Z\text{'}), (\text{`}A\text{'}, \text{`}A\text{'}), \ldots)\}$ be a function that maps each character to its upper-case representation. If a character does not have an upper-case representation, it is mapped to itself. We define the semantics of toupper as follows:

$$\text{toupper}(v) = t \Leftrightarrow \exists s_0, s_1, \ldots, s_i \in \Sigma\text{: } v = s_0 s_1 \ldots s_i \wedge t = f(s_0)f(s_1)\ldots f(s_i)$$

- tolower(v) transforms all characters of the string v into lower-case characters and returns the result. Let $f = \{(\text{`}a\text{'}, \text{`}a\text{'}), \ldots, (\text{`}A\text{'}, \text{`}a\text{'}), (\text{`}B\text{'}, \text{`}b\text{'}), \ldots, (\text{`}Z\text{'}, \text{`}z\text{'}), \ldots)\}$ be a function that maps each character to its lower-case representation. If a character

does not have an lower-case representation, it is mapped to itself. We define the semantics of tolower as follows:

$$\text{tolower}(v) = t \Leftrightarrow \exists s_0, s_1, \ldots, s_i \in \Sigma\colon v = s_0 s_1 \ldots s_i \wedge t = f(s_0) f(s_1) \ldots f(s_i)$$

- substring(v, i, n) returns the substring of the string v that starts at the index i (inclusive) and spans n characters (or until the end of the string, whichever comes first). We define the semantics of substring as follows:

$$\text{substring}(v, i, n) = t \Leftrightarrow \exists s_1, s_2 \in \Sigma^*\colon v = s_1 t s_2 \wedge |s_1| = i \wedge (|t| = n \vee s_2 = \varepsilon)$$

- replacefirst(v, p, t) finds the first substring of the string v that matches the regular expression p and replaces it with the target string t and returns the result. If more than one substring of string v matches the regular expression p at the same start index, the longest matching substring is chosen for replacement. If the regular expression p does not match any substring of the string v, string v is returned. We define the semantics of replacefirst as follows:

$$\begin{aligned}
&(\text{replacefirst}(v, p, t) = v \Leftrightarrow \forall s_1, s_2 \in \Sigma^*\colon \neg\text{match}(v, s_1 p s_2)) \wedge \\
&(\text{replacefirst}(v, p, t) = s \Leftrightarrow \exists m, s_1, s_2 \in \Sigma^*\colon v = s_1 m s_2 \wedge \text{match}(m, p) \wedge s = s_1 t s_2 \wedge \\
&\quad (s_1 = \varepsilon \vee (\forall s_3 \in \Sigma^*, s_4 \in \Sigma^+\colon s_1 = s_3 s_4 \wedge \neg\text{match}(s_3, p) \wedge \neg\text{match}(s_4 m, p))) \wedge \\
&\quad (s_2 = \varepsilon \vee (\forall s_5 \in \Sigma^+, s_6 \in \Sigma^*\colon s_2 = s_5 s_6 \wedge \neg\text{match}(m s_5, p))))
\end{aligned}$$

- replacelast(v, p, t) finds the last substring of the string v that matches the regular expression p and replaces it with the target string t and returns the result. If more than one substring of string v matches the regular expression p at the same start

index, the longest matching substring is chosen for replacement. If the regular expression p does not match any substring of the string v, string v is returned. We define the semantics of replacelast as follows:

$$(\text{replacelast}(v,p,t) = v \Leftrightarrow \forall s_1, s_2 \in \Sigma^* \colon \neg\text{match}(v, s_1 p s_2)) \wedge$$
$$(\text{replacelast}(v,p,t) = s \Leftrightarrow \exists m, s_1, s_2 \in \Sigma^* \colon v = s_1 m s_2 \wedge \text{match}(m,p) \wedge s = s_1 t s_2 \wedge$$
$$(s_1 = \varepsilon \vee (\forall s_3 \in \Sigma^*, s_4 \in \Sigma^+ \colon s_1 = s_3 s_4 \wedge \neg\text{match}(s_4 m, p))) \wedge$$
$$(s_2 = \varepsilon \vee (\forall s_5 \in \Sigma^+, s_6 \in \Sigma^* \colon s_2 = s_5 s_6 \wedge \neg\text{match}(s_6, p) \wedge \neg\text{match}(m s_5, p))))$$

- replaceall(v,p,t) replaces all substrings of the string v that matches the regular expression p with the target string t and returns the result. If more than one substring of the string v matches the regular expression at the same start index, the longest matching substring is chosen for replacement. If regular expression p does not match any substring of the string v, string v is returned. Let $\mathcal{L}(p)$ denote set of strings (i.e., the language) defined by the regular expression p, then we define

the semantics of replaceall as follows:

$$
\begin{aligned}
&(\text{replaceall}(v,p,t) = v \Leftrightarrow \forall s_1, s_2 \in \Sigma^* \colon \neg\text{match}(v, s_1 p s_2)) \wedge \\
&(\text{replaceall}(v,p,t) = t \Leftrightarrow v = \varepsilon \wedge \varepsilon \in \mathcal{L}(p)) \wedge \\
&(\text{replaceall}(v,p,t) = s \Leftrightarrow \exists s_1 \in \Sigma^+, s_2 \in \Sigma^* \colon v = s_1 s_2 \wedge \text{match}(s_1, p) \wedge \\
&\quad (s_2 = \varepsilon \vee (\forall s_3 \in \Sigma^+, s_4 \in \Sigma^* \colon s_2 = s_3 s_4 \wedge \neg\text{match}(m s_3, p))) \wedge \\
&\quad \exists s_5 \in \Sigma^* \colon s_5 = \text{replaceall}(s_2, p, t) \wedge s = t s_5) \wedge \\
&(\text{replaceall}(v,p,t) = s \Leftrightarrow \exists s_1 \in \Sigma, s_2 \in \Sigma^* \colon v = s_1 s_2 \wedge \\
&\quad (s_2 = \varepsilon \vee (\forall s_3 \in \Sigma^+, s_4 \in \Sigma^* \colon v = s_1 s_3 s_4 \wedge \neg\text{match}(s_1 s_3, p))) \wedge \\
&\quad \exists s_5 \in \Sigma^* \colon s_5 = \text{replaceall}(v_1, p, t) \wedge \\
&\quad ((\varepsilon \in \mathcal{L}(p) \wedge s = t s_1 s_5) \ \vee (\varepsilon \notin \mathcal{L}(p) \wedge s = s_1 s_5)))
\end{aligned}
$$

String operations that are not directly available in our constraint language can be defined using existing operations. As an example, we can define generic left trim and right trim operations as follows:

- ltrim(v, p) trims the characters that matches the regular expression p from the beginning of string v and returns the result. The semantics of ltrim can be defined as follows:

$$
\begin{aligned}
&(\text{ltrim}(v,p) = v \Leftrightarrow \neg\text{begins}(v,p)) \wedge \\
&(\text{ltrim}(v,p) = t \Leftrightarrow \text{begins}(v,p) \wedge t = \text{replacefirst}(s,p,\text{""}))
\end{aligned}
$$

- rtrim(v, p) trims the characters that matches the regular expression p from the end

of string v and returns the result. The semantics of rtrim can be defined as follows:

$$(\text{rtrim}(v, p) = v \Leftrightarrow \neg\text{ends}(s, p)) \wedge$$
$$(\text{rtrim}(v, p) = t \Leftrightarrow \text{ends}(v, p) \wedge t = \text{replacelast}(s, p, ""))$$

3.2 Mapping of Java String Expressions to the Constraint Language

JAVA is one of the most widely used programming language in modern software application development. It provides extensive support for string manipulation. JAVA standard libraries provide `String`, `StringBuffer`, `StringBuilder`, `CharSequence`, and `Character` classes that include string manipulation methods. String variables in our constraint language can be used to represent instances of these classes. Table 3.1 lists example mappings of JAVA string expressions to our constraint language.

Semantics of some of the string expressions in JAVA cannot be expressed precisely using the operations provided in our constraint language. For example, the translation for the JAVA expression `v.equalsIgnoreCase(t)` defined in Table 3.1 works precisely for an alphabet Σ (i.e., character encoding) with the following condition: $\forall c_1, c_2 \in \Sigma : c_1 = c_2 \Leftrightarrow (\text{toupper}(c_1) = \text{toupper}(c_2) \wedge \text{tolower}(c_1) = \text{tolower}(c_2))$. However, there can be character encodings in JAVA that does not hold the condition we defined for the translation of the `v.equalsIgnoreCase(t)`. More general cases can be handled by adding more specific operations into our language.

Java String Expression	**Constraint Language**
`s.length()`	$\text{length}(s)$
`s.isEmpty()`	$\text{length}(s) = 0$
`s.charAt(i)`	$\text{charat}(s, i)$
`s.equals(t)`	$s = t$
`s.equalsIgnoreCase(t)`	$s = t \vee \text{toupper}(s) = \text{toupper}(t) \vee \text{tolower}(s) = \text{tolower}(t)$
`s.startsWith(t)`	$\text{begins}(s, t)$
`s.startsWith(t,n)`	$0 \leq n \wedge n \leq \lvert s\rvert \wedge \text{begins}(\text{substring}(s, n, \lvert s\rvert - n), t)$
`s.endsWith(t)`	$\text{ends}(s, t)$
`s.indexOf(t)`	$\text{indexof}(s, t)$
`s.indexOf(t,n)`	$\text{indexof}(\text{substring}(s, n, \lvert s\rvert - n), t)$
`s.lastIndexOf(t)`	$\text{lastindexof}(s, t)$
`s.lastIndexOf(t,n)`	$\text{lastindexof}(\text{substring}(s, 0, n + 1), t)$
`s.substring(i)`	$\text{substring}(s, i, \lvert s\rvert)$
`s.substring(i,j)`	$\text{substring}(s, i, j - i)$
`s.concat(t)`	$s \cdot t$
`s.replace(t,r)`	$\text{replaceall}(s, t, r)$
`s.matches(p)`	$\text{match}(s, p)$
`s.contains(t)`	$\text{contains}(s, t)$
`s.replaceFirst(p,r)`	$\text{replacefirst}(s, p, r)$
`s.replaceAll(p,r)`	$\text{replaceall}(s, p, r)$
`join(s,t,w,y)`	$t \cdot s \cdot w \cdot s \cdot y$
`s.toLowerCase()`	$\text{tolower}(s)$
`s.toUpperCase()`	$\text{toupper}(s)$
`s.trim()`	$\text{ltrim}(\text{rtrim}(s, p \cdot p^*), p \cdot p^*)$
`valueOf(i)`	$\text{tostring}(i)$
`s.delete(i,j)`	$\text{substring}(s, 0, i) \cdot \text{substring}(s, j, \lvert s\rvert - j)$
`s.replace(i,j,t)`	$\text{substring}(s, 0, i) \cdot t \cdot \text{substring}(s, j, \lvert s\rvert - j)$
`s.insert(i,t)`	$\text{substring}(s, 0, i) \cdot t \cdot \text{substring}(s, i + 1, \lvert s\rvert - i - 1)$

Table 3.1: Examples of JAVA string expressions to constraint language translation. `s`, `r`, `t`, `w`, and `y` can be instances of `String`, `StringBuffer`, `StringBuilder`, `CharSequence`, and `Character` classes. `p` is a string that represents a valid regular expression. `b` is a boolean and `n`, `i`, and `j` are integers.

PHP String Expression	Constraint Language
`addslashes(s)`	replaceall(s, replaceall(s, "\", "\\"), "', "\'") ...
`chr(i)`	tostring(i)
`htmlspecialchars(s)`	replaceall(s, replaceall(s, "&", "&"), "<", "<") ...
`lcfirst(s)`	replacefirst(s, charat(s, 0), tolower(charat(s, 0))
`ltrim(s, p)`	ltrim($s, p \cdot p^*$)
`ord(s)`	toint(s)
`rtrim(s, p)`	rtrim($s, p \cdot p^*$)
`str_replace(p, t, s)`	replaceall(s, p, t)
`strcmp(s, t)`	$s = t$
`strlen(s)`	length(s)
`strpbrk(s, p)`	substring(s, indexof(s, t))
`strpos(s, t)`	indexof(s, t)
`strrpos(s, t)`	lastindexof(s, t)
`strtolower(s)`	tolower(s)
`strtoupper(s, t)`	toupper(s)
`substr_replace(s, t, i, j)`	substring($s, 0, i$) $\cdot$ t $\cdot$ substring($s, j, \|s\| - j$)
`substr(s, i, j)`	substring($s, i, j - i$)
`trim(s, p)`	trim(s, p)
`ucfirst(s)`	replacefirst(s, charat(s, 0), toupper(charat(s, 0))
`preg_replace(p, r, s)`	replaceall(s, p, r)

Table 3.2: PHP string expressions to constraint language translation. r, s, and t are string instances. p is a string that represents a valid regular expression. i and j are integers.

3.3 Mapping of PHP String Expressions to the Constraint Language

PHP is another popular web application development language. Since web applications do extensive string manipulation, PHP comes with a good support of string manipulation functions. Table 3.2 lists mappings of the PHP string expressions to our constraint language.

We have shown that our constraint language can handle a wide range of string expressions. The list of mappings we have can be extended to other programming languages that support string manipulation.

Chapter 4

Automata-based Constraint Solving

In this chapter, we discuss how to construct automata for string and integer constraints such that the constructed automata accept the set of solutions for the given constraints. We first introduce preliminary definitions and then we discuss the details of the automata-based constraint solving.

Given a formula φ, let $\varphi[s/v]$ denote the formula that is obtained from φ by replacing all appearances of variable $v \in \mathcal{V}(\varphi)$ with the constant s. We define the truth set of the formula φ for variable v as $[\![\varphi, v]\!] = \{s \mid \varphi[s/v] \text{ is satisfiable}\}$.

In our constraint solving approach, we map string and integer constraints to Deterministic Finite Automaton (DFA). A DFA A is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is the set of states, Σ is the input alphabet, $\delta\colon Q \times \Sigma \to Q$ is the state transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final, or accepting, states. Throughout the dissertation, whenever we refer to an automaton, we assume it is a DFA.

Given an automaton A, let $\mathcal{L}(A)$ denote the set of strings accepted by A. We define $\cap$ operation on automata such that $A_1 \cap A_2$ generates an automaton that accepts the language defined by $\mathcal{L}(A_1) \cap \mathcal{L}(A_2)$. Similarly, $\cup$ operation on automata, $A_1 \cup A_2$, generates an automaton that accepts the language defined by $\mathcal{L}(A_1) \cup \mathcal{L}(A_2)$. We define $\neg$

operation on automata such that $\neg A$ generates an automaton that accepts the language defined by $\Sigma^* \setminus \mathcal{L}(A)$. All three operations ($\cap$, $\cup$, $\neg$) can be implemented using automata product and automata complement operations.

Let $\overrightarrow{A}$ denote a dictionary such that, given a formula φ, $\overrightarrow{A} = \{v_1 \mapsto A_1, v_2 \mapsto A_2, \ldots, v_n \mapsto A_n\}$, where $v_1, v_2, \ldots, v_n \in \mathcal{V}(\varphi)$, is a mapping between variables and the corresponding automata. We denote the set of variables in a dictionary $\overrightarrow{A}$ as $\mathcal{V}(\overrightarrow{A})$. We define [] operator to access the elements of a dictionary, e.g., given a dictionary $\overrightarrow{A}$, $\overrightarrow{A}[v]$ returns the corresponding automaton from the dictionary. Let A_{Σ^*} denote an automaton that accepts any value, i.e., $\mathcal{L}(A_{\Sigma^*}) = \Sigma^*$. If a variable v is not contained in a dictionary $\overrightarrow{A}$, $\overrightarrow{A}[v]$ automatically inserts the mapping $v \mapsto A_{\Sigma^*}$ and returns a dictionary with the only mapping $v \mapsto A_{\Sigma^*}$ or returns the automaton A_{Σ^*} depending on the context.

Given a dictionary $\overrightarrow{A}$ and a set of variables V, $\overrightarrow{A}[V]$ returns a dictionary where it contains only the variables in the set V. If there are variables in V that do not have mappings in the dictionary $\overrightarrow{A}$, they are added to the returned dictionary and mapped to the A_{Σ^*} automaton.

Let $\overrightarrow{A}_1 = \{v_1 \mapsto A_{1v_1}, v_2 \mapsto A_{1v_2}, \ldots, v_n \mapsto A_{1v_n}\}$ and $\overrightarrow{A}_2 = \{v_1 \mapsto A_{2v_1}, v_2 \mapsto A_{2v_2}, \ldots, v_n \mapsto A_{2v_n}\}$. We extend $\cap$ operation such that $\overrightarrow{A}_1 \cap \overrightarrow{A}_2$ returns a dictionary $\overrightarrow{A}_\cap$ where $\overrightarrow{A}_\cap = \{v_1 \mapsto A_{1v_1} \cap A_{2v_1}, v_2 \mapsto A_{1v_2} \cap A_{2v_2}, \ldots, v_n \mapsto A_{1v_n} \cap A_{2v_n}\}$. Similarly, $\overrightarrow{A}_1 \cup \overrightarrow{A}_2$ returns a dictionary $\overrightarrow{A}_\cup$ where $\overrightarrow{A}_\cup = \{v_1 \mapsto A_{1v_1} \cup A_{2v_1}, v_2 \mapsto A_{1v_2} \cup A_{2v_2}, \ldots, v_n \mapsto A_{1v_n} \cup A_{2v_n}\}$.

Given a formula φ and a variable v, our goal is to construct an automaton A, such that $\mathcal{L}(A) = [\![\varphi, v]\!]$. If the set of variables $\mathcal{V}(\varphi)$ has more than one variable, we construct an automaton for each variable in $\mathcal{V}(\varphi)$. Let $\overrightarrow{A}$ represents the automata generated for each variable in $\mathcal{V}(\varphi)$. If $\exists v \in \mathcal{V}(\varphi) : \mathcal{L}(\overrightarrow{A}[v]) = \emptyset$, we say formula φ is unsatisfiable, otherwise, the formula might be satisfiable.

Note that, when there are multiple variables, one can specify constraints with non-

regular truth sets. For example, given the formula $\varphi \equiv x = y{\cdot}y$, $[\![\varphi, x]\!]$ is not a regular set, so we cannot construct an automaton precisely recognizing its truth set. In such cases, we generated an automaton A that over-approximates the truth set (i.e., $\mathcal{L}(A) \supseteq [\![\varphi, x]\!]$).

4.1 Mapping Formulae to Automata

Let us define an automata constructor function $\mathcal{A}$ (Algorithm 1) such that, given a formula φ, $\mathcal{A}(\varphi)$ generates an automaton for each variable v in $\mathcal{V}(\varphi)$ by traversing the syntax tree of the formula.

Some parts of our algorithms work by iteratively updating a set of *current solutions* for the variables, which we denote by $\vec{A}_{\text{context}}$. Before processing any part of the given constraint, any variable can take on any value, i.e., initially all variables are unconstrained. So given a formula φ, initially $\vec{A}_{\text{context}} = \{v_1 \mapsto A_{\Sigma^*}, v_2 \mapsto A_{\Sigma^*}, \ldots, v_n \mapsto A_{\Sigma^*}\}$ where $v_1, v_2, \ldots, v_n \in \mathcal{V}(\varphi)$.

Since the negation operator is not monotonic and since we sometimes over-approximate solution sets for subformulae, in line 3, we convert the given formula to negation normal form (by pushing negations to atomic formulas). Function $\mathcal{A}$ uses the automata constructor function $\mathcal{A}_\phi$ (line 6) to construct automata for atomic string and integer constraints, which is discussed in the following section. Function $\mathcal{A}_\phi$ computes sound over-approximations and can handle negated atomic constraints. It accepts an atomic constraint ($\varphi_{\mathbb{S}}$ or $\varphi_{\mathbb{Z}}$) and initial values for the variables appearing in the given formula as $\vec{A}_{\text{context}}$ and returns a solution automaton for each variable in the given atomic constraint.

Conjunctions and disjunctions are handled with $\cap$ and $\cup$ operations, respectively. In handling conjunctions, due to approximations, intersection may not sufficiently restrict the values of the variables since the relations among some string variables are lost during automata construction. In such cases, it is necessary to propagate the constraints on

Algorithm 1 Automata constructor function

Input: Formula φ: input formula
Output: Dictionary $\vec{A}$: an automaton for each variable in $\mathcal{V}(\varphi)$

1: **function** $\mathcal{A}(\varphi)$
2: **if** $\varphi \equiv \neg\varphi$ **then**
3: **return** $\mathcal{A}$(ToNegationNormalForm($\neg\varphi$))
4: **else if** $\varphi \equiv \varphi_{\mathbb{S}}$ **or** $\varphi \equiv \neg\varphi_{\mathbb{S}}$ **or** $\varphi \equiv \varphi_{\mathbb{Z}}$ **or** $\varphi \equiv \neg\varphi_{\mathbb{Z}}$ **then**
5: $\vec{A}_{\text{context}} \leftarrow \{v_1 \mapsto A_{\Sigma^*}, v_2 \mapsto A_{\Sigma^*}, \ldots, v_n \mapsto A_{\Sigma^*}\}$ where $v_1, v_2, \ldots, v_n \in \mathcal{V}(\varphi)$
6: **return** $\mathcal{A}_\phi(\varphi, \vec{A}_{\text{context}})$
7: **else if** $\varphi \equiv \varphi_1 \vee \varphi_2$ **then**
8: **return** $\mathcal{A}(\varphi_1)[\mathcal{V}(\varphi)] \cup \mathcal{A}(\varphi_2)[\mathcal{V}(\varphi)]$
9: **else if** $\varphi \equiv \varphi_1 \wedge \varphi_2$ **then**
10: **return** Refine($\varphi, \mathcal{A}(\varphi_1)[\mathcal{V}(\varphi)] \cap \mathcal{A}(\varphi_2)[\mathcal{V}(\varphi)]$)
11: **end if**
12: **end function**

a variable that is inferred by one side of the conjunction to the other side of the conjunction. We do this using the function Refine at line 10, which takes the input formula and the result automata of the intersection as parameters and returns automata which (possibly) provide a better approximations. It updates the automata iteratively on every intersection by solving the atomic constraints that may cause over-approximations (Algorithm 2).

Algorithm 2 Automata refinement

Input: Formula φ: a formula, Dictionary $\vec{A}$: an initial automaton for each variable in $\mathcal{V}(\varphi)$
Output: Dictionary $\vec{A}$: an updated automaton for each variable in $\mathcal{V}(\varphi)$

1: **function** Refine($\varphi, \vec{A}$)
2: **for each** $\phi \in$ {atomic sub constraint of φ with possible over-approximation} **do**
3: $\vec{A} \leftarrow \vec{A} \cap \mathcal{A}_\phi(\varphi_\phi, \vec{A}[\mathcal{V}(\varphi_\phi)])[\mathcal{V}(\varphi)]$
4: **end for**
5: **return** $\vec{A}$
6: **end function**

Let us consider the following illustrative example $\varphi \equiv \neg\text{match}(v, (\text{ab})^*) \wedge \text{length}(v) \geq 1$ over the alphabet $\Sigma = \{\text{a}, \text{b}\}$ before going into details of the automata constructor function $\mathcal{A}_\phi$ for atomic constraints.

Figure 4.1 shows syntax tree of the formula. Given the atomic constraint $\varphi_1 \equiv \neg\text{match}(v, (\text{ab})^*)$ and the initial value for the variable v as $\vec{A}_{\text{context}} = \{v \mapsto A_{\Sigma^*}\}$, au-

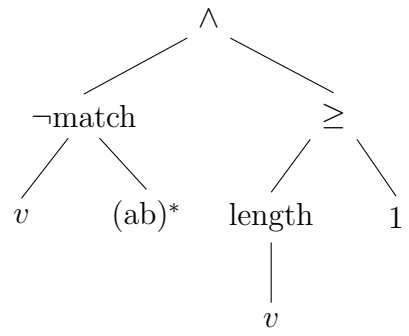

Figure 4.1: Syntax tree of the formula $\varphi \equiv \neg\text{match}(v, (\text{ab})^*) \wedge \text{length}(v) \geq 1$.

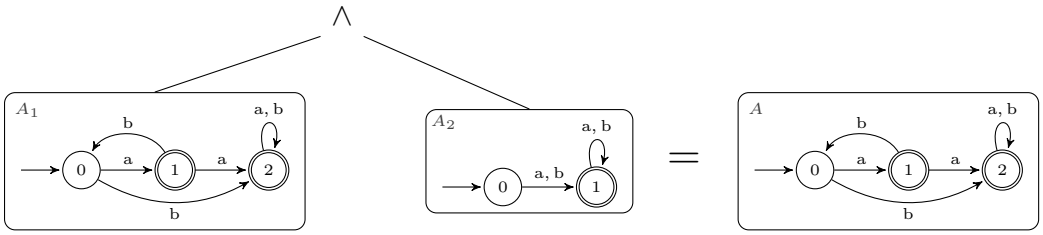

Figure 4.2: Automata constructed for the formula $\varphi \equiv \neg\text{match}(v, (\text{ab})^*) \wedge \text{length}(v) \geq 1$.

tomata constructor function $\mathcal{A}_\phi$ generates an automaton A_1 (Figure 4.2) for variable v and returns it as $\overrightarrow{A}_1 = \{v \mapsto A_1\}$. Next, given the atomic constraint $\varphi_2 \equiv \text{length}(v) \geq 1$ and the initial value for the variable v as $\overrightarrow{A}_{\text{context}} = \{v \mapsto A_{\Sigma^*}\}$, the function $\mathcal{A}_\phi$ generates an automaton A_2 (Figure 4.2) for variable v and returns it as $\overrightarrow{A}_2 = \{v \mapsto A_2\}$ Finally, conjunction is handled by intersecting automata generated for the atomic constraints ($\overrightarrow{A} = \overrightarrow{A}_1 \cap \overrightarrow{A}_2$) as shown in Figure 4.2. Since atomic constraints φ_1 and φ_2 are handled precisely, Refine function call does not result in any change on the final automaton computed for variable v.

Next, we will discuss the details of the automata construction for atomic constraints.

4.2 Handling Atomic Constraints

For a given atomic formula φ ($\varphi_{\mathbb{S}}$ or $\varphi_{\mathbb{Z}}$) and $\vec{A}_{\text{context}}$, automata constructor function $\mathcal{A}_\phi$ (Algorithm 3) constructs automata (a dictionary with automata) $\vec{A}$ where $\forall v \in \mathcal{V}(\varphi)$: $\mathcal{L}(\vec{A}[v]) \supseteq [\![\varphi, v]\!]$. Function $\mathcal{A}_\phi$ first constructs an automaton for each term appearing in the predicate operation ($\star$, where $\star \in \{=, <, >, \text{match}, \text{contains}, \text{begins}, \text{ends}\}$) using the function $\mathcal{A}_{\text{TermCons}}$ described in Algorithm 4. Then, it constructs an automaton for the predicate ($\star$) using the automata construction function $\mathcal{A}_\star$. We provide the details of the function $\mathcal{A}_\star$ in the next sub section. Parameters of a predicate operation can be any term (γ, β, ρ). Hence, the function $\mathcal{A}_\star$ introduces auxiliary variables that represents the parameters of the terms. Function $\mathcal{A}_\phi$ propagates the results of the string predicate operation to the variables appearing in the formula using the function $\mathcal{A}_{\text{TermProp}}$ described in Algorithm 5.

Algorithm 3 Automata constructor for atomic constraints

Input: Formula φ: an atomic constraint,
Dictionary $\vec{A}_{\text{context}}$: an initial automaton for each variable in $\mathcal{V}(\tau)$
Output: Dictionary $\vec{A}$: an updated automaton for each variable in $\mathcal{V}(\tau)$

function $\mathcal{A}_\phi(\varphi, \vec{A}_{\text{context}})$
  $\vec{A} \leftarrow \{\,\}$
  if $\varphi \equiv \tau_1 \star \tau_2$ **then** ▷ $\star \in \{=, <, >, \text{match}, \text{contains}, \text{begins}, \text{ends}\}$.
    $\vec{A} \leftarrow \mathcal{A}_\star(\mathcal{A}_{\text{TermCons}}(\tau_1, \vec{A}_{\text{context}}), \mathcal{A}_{\text{TermCons}}(\tau_2, \vec{A}_{\text{context}}))$
  else if $\varphi \equiv \neg(\gamma_1 \star \gamma_2)$ **then**
    $\vec{A} \leftarrow \mathcal{A}_{\bar{\star}}(\mathcal{A}_{\text{TermCons}}(\tau_1, \vec{A}_{\text{context}}), \mathcal{A}_{\text{TermCons}}(\tau_2, \vec{A}_{\text{context}}))$
  end if
  for each $(v, \tau) \in \{(a, b) \mid a \in \mathcal{V}(b) \wedge b \in \{\tau_1, \tau_2\} \wedge \varphi \equiv \tau_1 \star \tau_2\}$ **do**
    $\vec{A} \leftarrow \vec{A}[\mathcal{V}(\varphi)] \cap \mathcal{A}_{\text{TermProp}}(v, \tau, \vec{A}[v_\tau], \vec{A}_{\text{context}})[\mathcal{V}(\varphi)]$
  end for
  return $\vec{A}$
end function

Let us consider the atomic constraints $\varphi_1 \equiv \neg\text{match}(v, (\text{ab})^*)$ and $\varphi_2 \equiv \text{length}(v) \geq 1$ where the syntax tree is shown in Figure 4.1. Function $\mathcal{A}_\phi$ generates automata in Figure 4.3 for the terms in the atomic formulae. The algorithm first processes the formula φ_1 and then processes the formula φ_2. Initially, the variable v is unconstrained and term

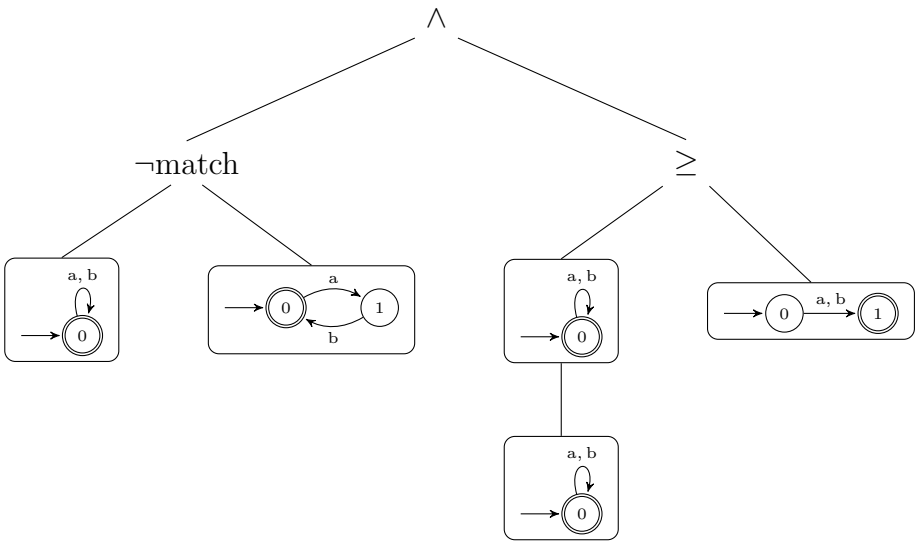

Figure 4.3: Automata construction for the terms in atomic the atomic formulae $\varphi_1 \equiv \neg\text{match}(v, (\text{ab})^*)$ and $\varphi_2 \equiv \text{length}(v) \geq 1$.

automaton constructor $\mathcal{A}_{\text{TermProp}}$ generates an automaton that accepts any string for the variable v in φ_1. An automaton that accepts the language of the regular expression is generated.

Given the automata generated for the parameters of the string predicate operation ¬match, the automata constructor function $\mathcal{A}_{\overline{\text{match}}}$ constructs an automaton for the first parameter of the string predicate operation (Figure 4.4).

Term propagation is necessary when we construct solution automaton for the term τ. This means that the relation between the term value and the values of variables in $\mathcal{V}(\tau)$ is lost (i.e., over-approximated) during term automata construction. Once an automaton for the predicate operation is constructed, $\mathcal{A}_{\text{TermProp}}$ propagates the result to each variable of $\mathcal{V}(\tau)$. Figure 4.3 shows automata constructed for each term using automata constructor function $\mathcal{A}_{\text{TermProp}}$. For the atomic constraint φ_1, it copies the result of the string predicate operation and associates it with the variable x.

Similarly, automata for the atomic constraint φ_2 is constructed (Figure 4.3), and after computing the automata based on the predicate operation $\geq$, the automaton generated

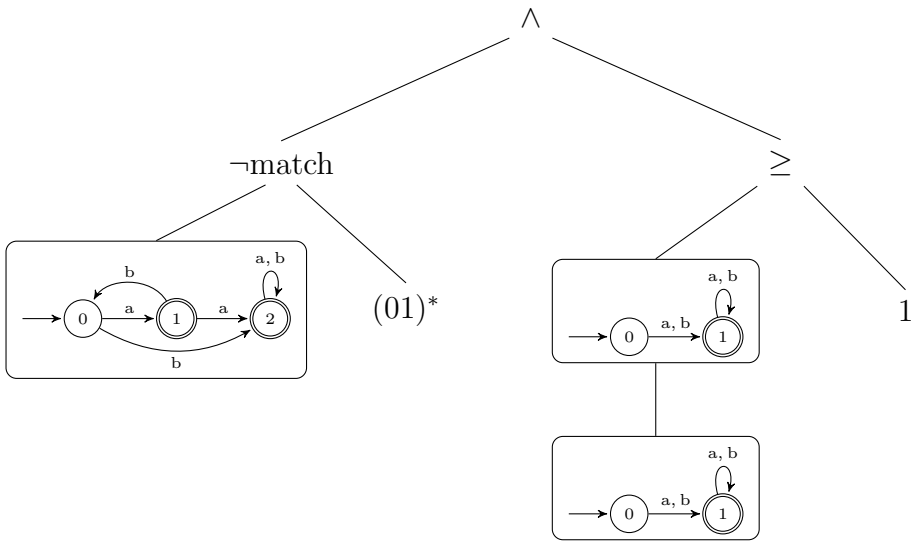

Figure 4.4: Automata propagation for the terms in atomic the atomic formulae $\varphi_1 \equiv \neg\text{match}(v, (\text{ab})^*)$ and $\varphi_2 \equiv \text{length}(v) \geq 1$.

for the length term is propagated back to the variable x (Figure 4.4).

We now discuss the functions $\mathcal{A}_{\text{TermCons}}$ and $\mathcal{A}_{\text{TermProp}}$ that are used during string automata construction. Given a term τ and $\overrightarrow{A}_{\text{context}}$, the function $\mathcal{A}_{\text{TermCons}}$ generates an automaton by recursive decomposition into subformulae (Algorithm 4). If τ is a regular expression term ρ, an automaton A_ρ is constructed using standard regular expression DFA constructions. If τ is an integer constant, A_n is constructed to recognize all strings of length n. For string variable terms, the current set of values for the variable is extracted from $\overrightarrow{A}_{\text{context}}$.

If τ corresponds to a term operation ($\odot \in \{\text{length}, \ldots, \text{replaceall}\}$), the corresponding term automata construction function $\mathcal{A}_\odot$ is called. The first parameter to the $\mathcal{A}_\odot$ functions indicates that we are constructing an automaton for the result of the term when it is set to 0. We also use the same function to construct automata for different arguments of a term (in which case the first parameter indicates the index of the argument) which is used in term propagation.

Now, we discuss the details of the function $\mathcal{A}_{\text{TermProp}}$. Given a variable v, a term

Algorithm 4 Term automata constructor

Input: Term τ : target term,
Dictionary $\overrightarrow{A}$: an initial automaton for each variable in $\mathcal{V}(\tau)$
Output: Automaton A_τ : an automaton computed based on τ operation

function $\mathcal{A}_{\text{TermCons}}(\tau, \overrightarrow{A}_{\text{context}})$
  if $\tau \equiv \rho$ **then**
    return A_ρ where $\mathcal{L}(A_\rho) = \{s \mid s \in \mathcal{L}(\rho)\}$.
  else if $\tau \equiv n$ **then**
    return A_n where $\mathcal{L}(A_n) = \{s \mid |s| = n\}$.
  else if $\tau \equiv v_s$ **then**
    return $\overrightarrow{A}_{\text{context}}[v_s]$
  else if $\tau \equiv v_i$ **then**
    return $\overrightarrow{A}_{\text{context}}[v_i]$
  else if $\tau \equiv \odot(\tau_1, \ldots, \tau_n)$ **then** ▷ $\odot \in \{\text{length}, \ldots, \text{replaceall}\}$.
    return $\mathcal{A}_\odot(0, \mathcal{A}_{\text{TermCons}}(\tau_1, \overrightarrow{A}_{\text{context}}), \ldots, \mathcal{A}_{\text{TermCons}}(\tau_n, \overrightarrow{A}_{\text{context}}))$
  end if
end function

τ that contains the variable v, an automaton A_τ that characterizes the values of the term τ, and $\overrightarrow{A}_{\text{context}}$, function $\mathcal{A}_{\text{TermProp}}$ recursively computes the values of the variable v that results in the values accepted by the automaton A_τ when the term operation that corresponds to τ is applied. When a term τ corresponds to a string variable, the updated set of values for the variable is computed by intersecting the propagated automaton A_τ with $\overrightarrow{A}_{\text{context}}$.

If a term τ corresponds to a term operation ($\odot \in \{\text{length}, \ldots, \text{replaceall}\}$), the corresponding term automata propagator function $\mathcal{A}_\odot$ is called to construct an automaton for the subterms that contain the variable v given the automata for the term τ and other subterms. The first parameter to the $\mathcal{A}_\odot$ functions indicates parameter index of the subterm that we propagate the values through. The next parameter is the automaton that corresponds to values of the result of the term operation that we want to propagate back. And the next parameters are the current values of the parameters to the term operation.

Algorithm 5 Term automata propagator

Input: Variable v: target variable, Term τ : current term, Automaton A_τ : result of term τ operation, Dictionary $\vec{A}$: an automaton for each variable in $\mathcal{V}(\tau)$
Output: Automaton A_v : an automaton for variable V

function $\mathcal{A}_{\text{TermProp}}(v, \tau, A_\tau, \vec{A}_{\text{context}})$
  if $\tau \equiv v_s$ **then**
    return $A_\tau \cap \vec{A}_{\text{context}}[v_s]$
  else if $\tau \equiv v_i$ **then**
    return $A_\tau \cap \vec{A}_{\text{context}}[v_i]$
  else if $\tau \equiv \odot(\tau_1, \ldots, \tau_n)$ **then** ▷ $\odot \in \{\text{length}, \ldots, \text{replaceall}\}$.
    for all $i \in \{k \mid 1 \leq k \leq n \wedge v \in \mathcal{V}(\tau_k)\}$ **do**
      $A_{\tau_i} \leftarrow \mathcal{A}_\odot(i, A_\tau, \mathcal{A}_{\text{TermCons}}(\tau_1, \vec{A}_{\text{context}}), \ldots, \mathcal{A}_{\text{TermCons}}(\tau_n, \vec{A}_{\text{context}}))$
      $\vec{A}_{\text{context}} \leftarrow \vec{A}_{\text{context}} \cap \mathcal{A}_{\text{TermProp}}(v, \tau_i, A_{\tau_i}, \vec{A}_{\text{context}})[\mathcal{V}(\vec{A}_{\text{context}})]$
    end for
    return $\vec{A}_{\text{context}}[v]$
  end if
end function

4.3 Automata Construction Semantics

For a given predicate operation $\star$ where $\star \in \{=, <, >, \text{match}, \text{contains}, \text{begins}, \text{ends}\}$ we use functions $\mathcal{A}_\star$ ($\mathcal{A}_{\bar{\star}}$ for negated predicate operations) to construct automata based on the predicate operation. For a given term operation $\odot$ where $\odot \in \{\text{length}, \text{toint}, \text{indexof}, \text{lastindexof}, \text{reverse}, \text{tostring}, \text{toupper}, \text{tolower}, \text{substring}, \text{replacefirst}, \text{replacelast}, \text{replaceall}\}$ we use functions $\mathcal{A}_\odot$ to construct automaton based on the term operation. In this section, we give automata construction semantics for each operation available in our constraint language.

4.3.1 Semantics for Predicate Operations

Let A_γ be an automaton generated for a string term and A_β be an automaton generated for a integer term. Automata constructor function $\mathcal{A}_\star$ takes two automata as input and refines them based on the predicate operation and returns the refined automata.

Let us define helper functions suffixes, prefixes, maxwords, minword, maxlengths, minlength that are used to define the semantics of automata constructions. Given an

automaton A, suffixes(A) returns an automaton that accepts all suffixes of all strings in $\mathcal{L}(A)$. Similarly, given an automaton A, prefixes(A) returns an automaton that accepts all prefixes of the all strings in $\mathcal{L}(A)$. Given a string s, suffixes(A) returns all suffixes of the string s where $\mathcal{L}(A) = \{s\}$. Similarly, given a string s, prefixes returns all prefixes of the string. Given an automaton A, maxwords returns an automaton that accepts lexicographically largest strings in $\mathcal{L}(A)$. Note that, there might be loops between states of the automaton A. For that reason, lexicographically largest strings can form an infinite set, which is again represented with an automaton that has loops. Given an automaton A, minword returns the lexicographically minimum string that is accepted by the automaton. Given an automaton A, maxlengths returns an automaton that accepts all the strings that has the same as lengths as the longest length strings in $\mathcal{L}(A)$ and minlength returns all the strings that has the same length with the smallest length string in $\mathcal{L}(A)$.

Table 4.1 shows the semantics of the automata constructions for the string predicate operations. String predicate operations takes two string terms (γ_1, γ_2) as parameters (match operation takes a string term and a regular expression term). Given automata A_{γ_1} and A_{γ_2} as parameters, each string predicate operation constructs automata A_1 and A_2 for the parameters and returns them in a dictionary with auxiliary variables v_{γ_1} and v_{γ_2} that correspond to the terms γ_1 and γ_2.

We define the semantics of the automata construction for the integer predicate operations as follows: Similar to the string predicate operations, integer predicate operations takes two integer terms (β_1, β_2) as parameters. Given automata A_{β_1} and A_{β_2} as parameters, each integer predicate operation constructs automata A_1 and A_2 for the parameters and returns them in a dictionary with auxiliary variables v_{β_1} and v_{β_2} that correspond to the terms β_1 and β_2. Table 4.2 shows the semantics of the automata constructions for the string predicate operations.

Operation	**Automata Construction Semantics**
$\mathcal{A}_{=}(A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A_1) = \mathcal{L}(A_2) = \{s \mid s \in \mathcal{L}(A_{\gamma_1}) \wedge s \in \mathcal{L}(A_{\gamma_2})$
$\mathcal{A}_{\neq}(A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A_1) = \{s \mid \exists t \in \mathcal{L}(A_{\gamma_2}) : s \neq t \wedge s \in \mathcal{L}(A_{\gamma_1})$ $\mathcal{L}(A_2) = \{s \mid \exists t \in \mathcal{L}(A_{\gamma_1}) : s \neq t \wedge s \in \mathcal{L}(A_{\gamma_2})$
$\mathcal{A}_{<}(A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A_1) = \{s \mid \exists t \in \mathcal{L}(\text{maxwords}(A_{\gamma_2})) : s \in \mathcal{L}(A_{\gamma_1}) \wedge s < t\}$ $\mathcal{L}(A_2) = \{s \mid s \in \mathcal{L}(A_{\gamma_2}) \wedge s > \text{minword}(A_{\gamma_1})$
$\mathcal{A}_{\leq}(A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A_1) = \{s \mid \exists t \in \mathcal{L}(\text{maxwords}(A_{\gamma_2})) : s \in \mathcal{L}(A_{\gamma_1}) \wedge s \leq t\}$ $\mathcal{L}(A_2) = \{s \mid s \in \mathcal{L}(A_{\gamma_2}) \wedge s \geq \text{minword}(A_{\gamma_1})\}$
$\mathcal{A}_{>}(A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A_1) = \{s \mid s \in \mathcal{L}(A_{\gamma_1}) \wedge s > \text{minword}(A_{\gamma_2})\}$ $\mathcal{L}(A_2) = \{s \mid \exists t \in \mathcal{L}(\text{maxwords}(A_{\gamma_1})) : s \in \mathcal{L}(A_{\gamma_2}) \wedge s < t\}$
$\mathcal{A}_{\geq}(A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A_1) = \{s \mid s \in \mathcal{L}(A_{\gamma_1}) \wedge s \geq \text{minword}(A_{\gamma_2})\}$ $\mathcal{L}(A_2) = \{s \mid \exists t \in \mathcal{L}(\text{maxwords}(A_{\gamma_1})) : s \in \mathcal{L}(A_{\gamma_2}) \wedge s \leq t\}$
$\mathcal{A}_{\text{match}}(A_{\gamma}, A_{\rho})$	$\mathcal{L}(A_1) = \{s \mid s \in \mathcal{L}(A_{\gamma}) \wedge s \in \mathcal{L}(A_{\rho})\}$ $\mathcal{L}(A_2) = \mathcal{L}(A_{\rho})$
$\mathcal{A}_{\overline{\text{match}}}(A_{\gamma}, A_{\rho})$	$\mathcal{L}(A_1) = \{s \mid s \in \mathcal{L}(A_{\gamma}) \wedge s \notin \mathcal{L}(A_{\rho})\}$ $\mathcal{L}(A_2) = \mathcal{L}(A_{\rho})$
$\mathcal{A}_{\text{begins}}(A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A_1) = \{s \mid s \in \mathcal{L}(A_{\gamma_1}) \wedge s \in \mathcal{L}(A_{\gamma_2})\Sigma^*\}$ $\mathcal{L}(A_2) = \{s \mid s \in \mathcal{L}(A_{\gamma_2}) \wedge s \in \mathcal{L}(\text{prefixes}(A_{\gamma_1}))\}$
$\mathcal{A}_{\overline{\text{begins}}}(A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A_1) = \{s \mid \exists t \in \mathcal{L}(A_{\gamma_2}) : s \in \mathcal{L}(A_{\gamma_1}) \wedge s \notin t\Sigma^*\}$ $\mathcal{L}(A_2) = \{s \mid \exists t \in \mathcal{L}(A_{\gamma_1}) : s \in \mathcal{L}(A_{\gamma_2}) \wedge s \notin \mathcal{L}(\text{prefixes}(t))\}$
$\mathcal{A}_{\text{ends}}(A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A_1) = \{s \mid s \in \mathcal{L}(A_{\gamma_1}) \wedge s \in \Sigma^*\mathcal{L}(A_{\gamma_2})\}$ $\mathcal{L}(A_2) = \{s \mid s \in \mathcal{L}(A_{\gamma_2}) \wedge s \in \mathcal{L}(\text{suffixes}(A_{\gamma_1}))\}$
$\mathcal{A}_{\overline{\text{ends}}}(A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A_1) = \{s \mid \exists t \in \mathcal{L}(A_{\gamma_2}) : s \in \mathcal{L}(A_{\gamma_1}) \wedge s \notin \Sigma^*t\}$ $\mathcal{L}(A_2) = \{s \mid \exists t \in \mathcal{L}(A_{\gamma_1}) : s \in \mathcal{L}(A_{\gamma_2}) \wedge s \notin \mathcal{L}(\text{suffixes}(t))\}$
$\mathcal{A}_{\text{contains}}(A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A_1) = \{s \mid s \in \mathcal{L}(A_{\gamma_1}) \wedge s \in \Sigma^*\mathcal{L}(A_{\gamma_2})\Sigma^*\}$ $\mathcal{L}(A_2) = \{s \mid s \in \mathcal{L}(A_{\gamma_2}) \wedge s \in \mathcal{L}(\text{suffixes}(\text{prefixes}(A_{\gamma_1})))\}$
$\mathcal{A}_{\overline{\text{contains}}}(A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A_1) = \{s \mid \exists t \in \mathcal{L}(A_{\gamma_2}) : s \in \mathcal{L}(A_{\gamma_1}) \wedge s \notin \Sigma^*t\Sigma^*\}$ $\mathcal{L}(A_2) = \{s \mid \exists t \in \mathcal{L}(A_{\gamma_1}) : s \in \mathcal{L}(A_{\gamma_2}) \wedge s \notin \mathcal{L}(\text{suffixes}(\text{prefixes}(t)))\}$

Table 4.1: Semantics of the string predicate automata constructors $\mathcal{A}_{\star}$ and $\mathcal{A}_{\overline{\star}}$. Operations returns a dictionary $\{v_{\gamma_1} \mapsto A_1, v_{\gamma_2} \mapsto A_2\}$ where A_1 and A_2 are automata constructed for terms, v_{γ_1} and v_{γ_2} are auxiliary variables.

Operation	**Automata Construction Semantics**
$\mathcal{A}_{=}(A_{\beta_1}, A_{\beta_2})$	$\mathcal{L}(A_1) = \mathcal{L}(A_2) = \{s \mid \exists t \in \Sigma^* : \lvert s\rvert = \lvert t\rvert \wedge t \in \mathcal{L}(A_{\beta_1}) \wedge t \in \mathcal{L}(A_{\beta_2})\}$
$\mathcal{A}_{\neq}(A_{\beta_1}, A_{\beta_2})$	$\mathcal{L}(A_1) = \{s \mid \exists v \in \Sigma^*, t \in \mathcal{L}(A_{\beta_2}) : \lvert s\rvert = \lvert v\rvert \wedge v \neq t \wedge v \in \mathcal{L}(A_{\beta_1})\}$ $\mathcal{L}(A_2) = \{s \mid \exists v \in \Sigma^*, t \in \mathcal{L}(A_{\beta_1}) : \lvert s\rvert = \lvert v\rvert \wedge v \neq t \wedge v \in \mathcal{L}(A_{\beta_2})\}$
$\mathcal{A}_{<}(A_{\beta_1}, A_{\beta_2})$	$\mathcal{L}(A_1) = \{s \mid \exists v \in \Sigma^*, t \in \mathcal{L}(\text{maxlengths}(A_{\beta_2})) : \lvert s\rvert = \lvert v\rvert \wedge v \in \mathcal{L}(A_{\beta_1}) \wedge \lvert v\rvert < \lvert t\rvert\}$ $\mathcal{L}(A_2) = \{s \mid \exists v \in \Sigma^* : \lvert s\rvert = \lvert v\rvert \wedge v \in \mathcal{L}(A_{\beta_2}) \wedge \lvert v\rvert > \lvert\text{minlength}(A_{\beta_1})\rvert\}$
$\mathcal{A}_{\leq}(A_{\beta_1}, A_{\beta_2})$	$\mathcal{L}(A_1) = \{s \mid \exists v \in \Sigma^*, t \in \mathcal{L}(\text{maxlengths}(A_{\beta_2})) : \lvert s\rvert = \lvert v\rvert \wedge v \in \mathcal{L}(A_{\beta_1}) \wedge \lvert v\rvert \leq \lvert t\rvert\}$ $\mathcal{L}(A_2) = \{s \mid \exists v \in \Sigma^* : \lvert s\rvert = \lvert v\rvert \wedge v \in \mathcal{L}(A_{\beta_2}) \wedge \lvert v\rvert \geq \lvert\text{minlength}(A_{\beta_1})\rvert\}$
$\mathcal{A}_{>}(A_{\beta_1}, A_{\beta_2})$	$\mathcal{L}(A_1) = \{s \mid \exists v \in \Sigma^* : \lvert s\rvert = \lvert v\rvert \wedge v \in \mathcal{L}(A_{\beta_1}) \wedge \lvert v\rvert > \lvert\text{minlength}(A_{\beta_2})\rvert\}$ $\mathcal{L}(A_2) = \{s \mid \exists v \in \Sigma^*, t \in \mathcal{L}(\text{maxlengths}(A_{\beta_1})) : \lvert s\rvert = \lvert v\rvert \wedge v \in \mathcal{L}(A_{\beta_2}) \wedge \lvert v\rvert < \lvert t\rvert\}$
$\mathcal{A}_{\geq}(A_{\beta_1}, A_{\beta_2})$	$\mathcal{L}(A_1) = \{s \mid \exists v \in \Sigma^* : \lvert s\rvert = \lvert v\rvert \wedge v \in \mathcal{L}(A_{\beta_1}) \wedge \lvert v\rvert \geq \lvert\text{minlength}(A_{\beta_2})\rvert\}$ $\mathcal{L}(A_2) = \{s \mid \exists v \in \Sigma^*, t \in \mathcal{L}(\text{maxlengths}(A_{\beta_1})) : \lvert s\rvert = \lvert v\rvert \wedge v \in \mathcal{L}(A_{\beta_2}) \wedge \lvert v\rvert \leq \lvert t\rvert\}$

Table 4.2: Semantics of the integer predicate automata constructors $\mathcal{A}_{\star}$ and $\mathcal{A}_{\bar{\star}}$. Operations returns a dictionary $\{v_{\beta_1} \mapsto A_1, v_{\beta_2} \mapsto A_2\}$ where A_1 and A_2 are automata constructed for terms, v_{β_1} and v_{β_2} are auxiliary variables.

4.3.2 Semantics for Term Operations

Functions $\mathcal{A}_{\odot}$ are used to generate automata for term operations. The first parameter to the $\mathcal{A}_{\odot}$ functions indicates that we are constructing an automaton for the result of the term when it is set to 0. We also use the same function to construct automata for different parameters of a term (in which case the first parameter indicates the index of the argument) which is used in term propagation. In the case of term propagation, we provide the result of the operation as second parameter. Informally, term propagation constructs and automaton for a parameter of the term operation, given the result of the term operation and the initial values of the parameters of the term operation. For example, lets consider the term operation $\tau \equiv \text{length}(A_\gamma)$. Given parameter index 1, an automaton A_{length} that corresponds to the expected results of the term operation τ, and an automaton A_γ for the initial values of the parameter; the function call $\mathcal{A}_{\text{length}}(1, A_{\text{length}}, A_\gamma)$ computes the values of the input parameter at index 1 (there is only one input parameter for length

Operation	**Automata Construction Semantics**
$\mathcal{A}_{\text{length}}(0, A_\gamma)$	$\mathcal{L}(A) = \{s \mid \exists t \in \mathcal{L}(A_\gamma) : \lvert s\rvert = \lvert t\rvert\}$
$\mathcal{A}_{\text{indexof}}(0, A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A) = \{s \mid \exists t \in \mathcal{L}(\text{prefixes}(A_{\gamma_1})), u \in \mathcal{L}(A_{\gamma_2}), v \in \Sigma^* : tuv \in \mathcal{L}(A_{\gamma_1}) \wedge \nexists t_1 \in \mathcal{L}(\text{suffixes}(\text{prefixes}(t))) : t_1 = u \wedge \lvert s\rvert = \lvert t\rvert$
$\mathcal{A}_{\text{lastindexof}}(0, A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A) = \{s \mid \exists t \in \mathcal{L}(\text{suffixes}(A_{\gamma_1})), u \in \mathcal{L}(A_{\gamma_2}), v \in \Sigma^* : vut \in \mathcal{L}(A_{\gamma_1}) \wedge \nexists t_1 \in \mathcal{L}(\text{suffixes}(\text{prefixes}(t))) : t_1 = u \wedge \lvert s\rvert = \lvert v\rvert$
$\mathcal{A}_{\text{substring}}(0, A_\gamma, A_{\beta_1}, A_{\beta_2})$	$\mathcal{L}(A) = \{s \mid \exists t \in \mathcal{L}(A_\gamma) : \exists t_1 \in \mathcal{L}(\text{prefixes}(t)), t_2 \in \Sigma^* : t = t_1 t_2 \wedge \lvert t_1\rvert \in \mathcal{L}(A_{\beta_1}) \wedge \exists v \in \mathcal{L}(\text{suffixes}(t_2)) : \lvert v\rvert \in \mathcal{L}(A_{\beta_2}) \wedge s = v\}$
$\mathcal{A}_{\text{charat}}(0, A_\gamma, A_\beta)$	$\mathcal{L}(A) = \{s \mid \exists t \in \mathcal{L}(A_\gamma) : \exists t_1 \in \mathcal{L}(\text{prefixes}(t)), t_2 \in \Sigma^* : t = t_1 t_2 \wedge \lvert t_1\rvert \in \mathcal{L}(A_{\beta_1}) \wedge \exists v \in \mathcal{L}(\text{suffixes}(t_2)) : \lvert v\rvert = 1 \wedge s = v\}$

Table 4.3: Example semantics of the term automata constructions for the result of a term. Term operations return an automaton A as a result.

term) that results in values in $\mathcal{L}(A_{\text{length}})$ when length operation is applied.

Table 4.3 and Table 4.4 show examples of automata construction semantics for the term operations. These constructions are based on pre- and post-image computations in string analysis similar to the ones used in [29, 13, 5].

We have shown automata construction details for the variables that appear in a given formula. We provided algorithms to handle boolean connectives and atomic constraints with complex term operations. The techniques described in this chapter generate an automaton for each variable. If there are multiple variables in a formula (i.e., relational constraints in a formula), the automata construction techniques described here cannot keep the relations precisely. In the next chapter, we discuss how to improve automata construction for relational constraints in order to present the set of solutions with better precision.

Operation	**Automata Construction Semantics**
$\mathcal{A}_{\text{length}}(1, A_{\text{length}}, A_\gamma)$	$\mathcal{L}(A) = \{s \mid s \in \mathcal{L}(A_\gamma) \wedge \exists t \in \mathcal{L}(A_{\text{length}}) : \lvert s\rvert = \lvert t\rvert\}$
$\mathcal{A}_{\text{indexof}}(1, A_{\text{indexof}}, A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A) = \{s \mid s \in \mathcal{L}(A_{\gamma_1}) \wedge \exists t, u, v \in \Sigma^* : t \in \mathcal{L}(A_{\text{indexof}}) \wedge u \in \mathcal{L}(A_{\gamma_2}) \wedge s = tuv\}$
$\mathcal{A}_{\text{indexof}}(2, A_{\text{indexof}}, A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A) = \{s \mid s \in \mathcal{L}(A_{\gamma_2}) \wedge \exists t, v \in \Sigma^* : t \in \mathcal{L}(A_{\text{indexof}}) \wedge tsv \in \mathcal{L}(A_{\gamma_1})\}$
$\mathcal{A}_{\text{lastindexof}}(1, A_{\text{lastindexof}}, A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A) = \{s \mid s \in \mathcal{L}(A_{\gamma_1}) \wedge \exists t, u, v \in \Sigma^* : t \in \mathcal{L}(A_{\text{lastindexof}}) \wedge u \in \mathcal{L}(A_{\gamma_2}) \wedge s = tuv\}$
$\mathcal{A}_{\text{lastindexof}}(2, A_{\text{lastindexof}}, A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A) = \{s \mid s \in \mathcal{L}(A_{\gamma_2}) \wedge \exists t, v \in \Sigma^* : t \in \mathcal{L}(A_{\text{lastindexof}}) \wedge tsv \in \mathcal{L}(A_{\gamma_1})\}$
$\mathcal{A}_{\text{substring}}(1, A_{\text{substring}}, A_\gamma, A_{\beta_1}, A_{\beta_2})$	$\mathcal{L}(A) = \{s \mid s \in \mathcal{L}(A_\gamma) \wedge \exists t, u \in \Sigma^*, v \in \mathcal{L}(A_{\text{substring}}) : \lvert t\rvert \in \mathcal{L}(A_{\beta_1}) \wedge \lvert v\rvert \in \mathcal{L}(A_{\beta_2}) \wedge s = tvu\}$
$\mathcal{A}_{\text{substring}}(2, A_{\text{substring}}, A_\gamma, A_{\beta_1}, A_{\beta_2})$	$\mathcal{L}(A) = \{s \mid s \in \mathcal{L}(A_{\beta_1}) \wedge \exists t, u \in \Sigma^*, s_1 \in \mathcal{L}(A_\gamma), v \in \mathcal{L}(A_{\text{substring}}) : \lvert v\rvert \in \mathcal{L}(A_{\beta_2}) \wedge s_1 = tvu \wedge \lvert s\rvert = \lvert t\rvert\}$
$\mathcal{A}_{\text{substring}}(3, A_{\text{substring}}, A_\gamma, A_{\beta_1}, A_{\beta_2})$	$\mathcal{L}(A) = \{s \mid s \in \mathcal{L}(A_{\beta_2}) \wedge \exists t, u \in \Sigma^*, s_1 \in \mathcal{L}(A_\gamma), v \in \mathcal{L}(A_{\text{substring}}) : \lvert t\rvert \in \mathcal{L}(A_{\beta_1}) \wedge s_1 = tvu \wedge \lvert s\rvert = \lvert v\rvert\}$
$\mathcal{A}_{\text{charat}}(1, A_{\text{charat}}, A_\gamma, A_\beta)$	$\mathcal{L}(A) = \{s \mid s \in \mathcal{L}(A_\gamma) \wedge \exists t, u \in \Sigma^*, v \in \mathcal{L}(A_{\text{charat}}) : \lvert t\rvert \in \mathcal{L}(A_\beta) \wedge s = tvu\}$
$\mathcal{A}_{\text{charat}}(2, A_{\text{charat}}, A_\gamma, A_\beta)$	$\mathcal{L}(A) = \{s \mid s \in \mathcal{L}(A_\beta) \wedge \exists t, u \in \Sigma^*, s_1 \in \mathcal{L}(A_\gamma), v \in \mathcal{L}(A_{\text{charat}}) : s_1 = tvu \wedge \lvert s\rvert = \lvert t\rvert\}$

Table 4.4: Example semantics of the term automata constructions for the parameters of a term. Term operations return an automaton A as a result.

Chapter 5

Relational Constraint Solving

In this chapter, we improve the precision of automata-based constraint solving techniques by better handling relations between multiple variables. In the previous chapter, we described algorithms that, given a formula, generate an automaton for each variable that characterizes the set of solutions for that variable. Since, in that approach, each variable has its own automaton, the relationships among variables are lost during automata construction. A multi-track DFA, which we define below, is a generalization of a DFA that accepts tuples of strings. Hence, a multi-track DFA can represent relations among different variables. We enhance the algorithms discussed in the previous chapter by using multi-track DFA in order to obtain better precision in constraint solving. We first introduce several definitions and then we revisit the automata construction algorithms using multi-track DFA.

A multi-track DFA A is a 5-tuple $(Q, (\Sigma \cup \{\lambda\})^k, \delta, q_0, F)$, where Q is the set of states, $\vec{\Sigma} = (\Sigma \cup \{\lambda\})^k$ is the k-track input alphabet, where $\lambda \notin \Sigma$ is a special padding symbol that appears only at the end of a string in each track, $\delta \colon Q \times \vec{\Sigma} \to Q$ is the transition relation, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of accepting states. Multi-track DFAs are closed under intersection, union and complement [43]. We associate with each

track of A a unique identifier v_i, which we refer to as the variable for track i. The set of track variables for A is denoted as $\mathcal{V}(A)$. The language of all strings recognized by A is denoted as $\mathcal{L}(A)$ where $\mathcal{L}(A) \subseteq \vec{\Sigma}^*$. Given a word $w \in \mathcal{L}(A)$, we use $w[v_i] \in \Sigma^*$ to denote the value of track i. Hence, $w \in \mathcal{L}(A)$ denotes a tuple of values $(w[v_1], w[v_2], \ldots, w[v_k])$, one value for each variable in $\mathcal{V}(A)$.

Given a formula φ, our goal is to construct an automaton A, such that $\mathcal{L}(A) = [\![\varphi]\!]$, where the tracks of A correspond to the variables of φ. We call such a DFA the *solution automaton* for φ. As some mixed constraints and even some pure string constraints admit non-regular truth sets [43], we provide a sound over approximation as an automaton A such that $[\![\varphi]\!] \subseteq \mathcal{L}(A)$ when exact truth sets are not precisely representable with DFA.

As we shall see, our method relies on carefully combining solution automata for subformulas with possibly different variable sets. Doing so requires that automata which are to be combined have the same tracks (sets of variable labels). Hence, we define an operation $[\,]$ such that, given an automaton A and a variable set V, $A[V]$ is an automaton A' where $\mathcal{V}(A') = V$. Let $x_1, \ldots, x_n \in V \setminus \mathcal{V}(A)$ be the variables in V but not in $\mathcal{V}(A)$ and $y_1, \ldots, y_m \in \mathcal{V}(A) \setminus V$ be the variables in $\mathcal{V}(A)$ but not in V. That is, we wish to add new unconstrained x_i tracks to A and remove y_i tracks from A. Then, we define $A[V]$ to be a multi-track DFA A' with $\mathcal{V}(A') = V$ such that $w' \in \mathcal{L}(A') \iff \exists w \in \mathcal{L}(A), \forall v \in \mathcal{V}(A') \cap \mathcal{V}(A), w[v] = w'[v]$.

Finally, some parts of our algorithms work by iteratively updating a set of *current solutions* for the variables, which we denote by A_{context}. Before processing any part of the given constraint, any variable can take any value, so initially $\mathcal{L}(A_{\text{context}}) = \vec{\Sigma}^*$, and we write $A_{\vec{\Sigma}^*}$ to indicate such an unconstrained automaton.

Note that a multi-track automaton is equivalent to a single track automaton when there is only one track. We use the same notation for both automata (A) in the rest of the dissertation. We now revisit the automata construction algorithms, this time using

multi-track automata instead of using a dictionary of single track automata.

Given a formula φ, the automata constructor function $\mathcal{A}$ (Algorithm 6) recursively constructs automata for sub-formulae of φ, combining them to return an automaton A such that $[\![\varphi]\!] \subseteq \mathcal{L}(A)$.

Function $\mathcal{A}$ uses the string automata constructor function $\mathcal{A}_{\mathbb{S}}$ (line 5) and integer automata constructor function $\mathcal{A}_{\mathbb{Z}}$ (line 7) to construct automata for atomic string and integer constraints, which are discussed in the following sections. Both $\mathcal{A}_{\mathbb{S}}$ and $\mathcal{A}_{\mathbb{Z}}$ compute sound over-approximations and can handle negated atomic constraints. Both functions accept an atomic constraint ($\varphi_{\mathbb{S}}$ or $\varphi_{\mathbb{Z}}$) and an automaton A_{context} (initially $A_{\vec{\Sigma}^*}$) and return a solution automaton that encodes the relations between variables in the given formula.

Conjunctions and disjunctions are handled with automata intersection and union (implemented using automata product operation) as described in the previous chapter. In both cases, we apply [ ] operator in order to ensure that variable tracks are compatible for the automata which are to be combined. We still need to apply function Refine at line 11 as there might still be over-approximations.

Algorithm 6 Relational automata constructor function

Input: Formula φ: input formula
Output: Multi-track DFA A : a solution automaton

1: **function** $\mathcal{A}(\varphi)$
2: **if** $\varphi \equiv \neg\varphi$ **then**
3: **return** $\mathcal{A}(\text{ToNegationNormalForm}(\neg\varphi))$
4: **else if** $\varphi \equiv \varphi_{\mathbb{S}}$ **or** $\varphi \equiv \neg\varphi_{\mathbb{S}}$ **then**
5: **return** $\mathcal{A}_{\mathbb{S}}(\varphi, A_{\vec{\Sigma}^*})$
6: **else if** $\varphi \equiv \varphi_{\mathbb{Z}}$ **or** $\varphi \equiv \neg\varphi_{\mathbb{Z}}$ **then**
7: **return** $\mathcal{A}_{\mathbb{Z}}(\varphi, A_{\vec{\Sigma}^*})$
8: **else if** $\varphi \equiv \varphi_1 \vee \varphi_2$ **then**
9: **return** $\mathcal{A}(\varphi_1)[\mathcal{V}(\varphi)] \cup \mathcal{A}(\varphi_2)[\mathcal{V}(\varphi)]$
10: **else if** $\varphi \equiv \varphi_1 \wedge \varphi_2$ **then**
11: **return** $\text{Refine}(\varphi, \mathcal{A}(\varphi_1)[\mathcal{V}(\varphi)] \cap \mathcal{A}(\varphi_2)[\mathcal{V}(\varphi)])$
12: **end if**
13: **end function**

Algorithm 7 Relational automata refinement

Input: Formula φ: a formula, Automaton A : initial values of the variables where $\mathcal{V}(A) = \mathcal{V}(\varphi)$
Output: Automaton A : updated values of the variables

```
1: function Refine(φ, A)
2:     for each φ ∈ {atomic sub constraint of φ with possible over-approximation} do
3:         A ← A ∩ A_φ(φ_φ, A[V(φ_φ)])[V(φ)]
4:     end for
5:     return A
6: end function
```

Examples. Let us consider the following examples:

$$\text{charat}(v, i) = \texttt{"a"} \wedge i = 2 \times j \tag{5.1}$$

$$\text{begins}(v, t) \wedge x = \text{length}(v) + 1 \tag{5.2}$$

Example 5.1 is a conjunction of an atomic string constraint $\varphi_1 \equiv \text{charat}(v, i) = \texttt{"a"}$ and a numeric constraint $\varphi_2 \equiv i = 2 \times j$. String constraint φ_1 is also a mixed constraint as it contains both a string and an integer variable. Given the string formula φ_1 and $A_{\text{context}} = A_{\vec{\Sigma}^*}$, string automata constructor $\mathcal{A}_{\mathbb{S}}$ constructs an automaton A_1 where $\mathcal{V}(A_1) = \{v, i, c_v\}$ where c_v is an auxiliary variable added for charat term appearing in the string predicate $=$. An example accepting tuple for the automaton A_1 is $(\texttt{"bab"}, 1, \texttt{"a"})$. Similarly, $\mathcal{A}_{\mathbb{Z}}$ constructs an automaton A_2 where $\mathcal{V}(A_2) = \{i, j\}$. An example accepting tuple for the automaton A_2 is $(-2, -1)$. Function π extends both automata A_1 and A_2 to the variable set $\{v, i, j, c_v\}$. Integer variable i appears on both constraints. Intersection of the both automata generates a new automaton A. As a result of the intersection, variable i represents non-negative even integers and variable j represents non-negative integers. However, intersection does not refine the values of the variables v and c_v as the automaton A_1 cannot encode the relation between string variable v and integer variable i. An example accepting tuple of the intersection automaton is $(\texttt{"bab"}, 2, 1, \texttt{"b"})$.

Function Refine executes the string automata constructor function $\mathcal{A}_{\mathbb{S}}$ once more for the mixed string constraint φ_1, this time by passing the automaton that represents the current values of the variables (A_{context}) as second parameter to refine the values of the string variable v based on the values of the integer variable i. An example accepting tuple of the final refined automaton is (`"aba"`, 2, 1, `"a"`).

Example 5.2 is handled in the same way Example 5.1 is handled. The only difference is that Example 5.2 contains a mixed integer constraint whereas Example 5.1 contains a mixed string constraint. Automata constructor function $\mathcal{A}$ constructs an automaton A where $\mathcal{V}(A) = \{v, t, x, l_v\}$. Integer automata constructor function adds auxiliary variable l_v to represent values of the length. An example accepting tuple for the resulting automaton is (`"ab"`, `"a"`, 3, 2).

Next, we discuss the algorithmic details of the functions $\mathcal{A}_{\mathbb{S}}$, $\mathcal{A}_{\mathbb{Z}}$, and how we handle the mixed constraints.

5.1 String Constraint Solving

For a given atomic string formula $\varphi_{\mathbb{S}}$ and A_{context}, string automata constructor function $\mathcal{A}_{\mathbb{S}}$ (Algorithm 8) constructs a multi-track DFA A where $[\![\varphi_{\mathbb{S}}]\!] \subseteq \mathcal{L}(A)$. Function $\mathcal{A}_{\mathbb{S}}$ first constructs an automaton for each string term appearing in string predicate ($\star$, where $\star \in \{=, <, >, \text{match}, \text{contains}, \text{begins}, \text{ends}\}$) using the function $\mathcal{A}_{\text{TermCons}}$ described in Algorithm 9. Then, it constructs an automaton for the string predicate ($\star$) using the corresponding automaton construction function. We provide the semantics of the relational automata constructions for string predicate operations which is the one of the main improvements over the Algorithm 3. And finally, if necessary, it propagates the result of the string predicate operation to the variables appearing in the formula using the function $\mathcal{A}_{\text{TermProp}}$ described in Algorithm 10.

Term propagation is necessary when we construct a single-track solution automaton for the term τ. This means that the relation between the term value and the values of variables in $\mathcal{V}(\tau)$ is lost (i.e., over-approximated) during term automata construction. Once the multi-track automaton for $\varphi_{\mathbb{S}}$ or $\varphi_{\mathbb{Z}}$ is constructed, $\mathcal{A}_{\text{TermProp}}$ propagates the result to each variable of $\mathcal{V}(\tau)$.

Algorithm 8 Relational string automata constructor

Input: Formula φ: an atomic constraint, Automaton A_{context} : initial values of the variables where $\mathcal{V}(A) = \mathcal{V}(\varphi)$
Output: Automaton $A_{\mathbb{S}}$: updated values of the variables

1: **function** $\mathcal{A}_{\mathbb{S}}(\varphi, A_{\text{context}})$
2: **if** $\varphi \equiv \gamma_1 \star \gamma_2$ **then** $\triangleright \star \in \{=, <, >, \text{match}, \text{contains}, \text{begins}, \text{ends}\}$.
3: $A_{\mathbb{S}} \leftarrow \mathcal{A}_{\star}(\mathcal{A}_{\text{TermCons}}(\gamma_1, A_{\text{context}}), \mathcal{A}_{\text{TermCons}}(\gamma_2, A_{\text{context}}))$
4: **else if** $\varphi \equiv \neg(\gamma_1 \star \gamma_2)$ **then**
5: $A_{\mathbb{S}} \leftarrow \mathcal{A}_{\overline{\star}}(\mathcal{A}_{\text{TermCons}}(\gamma_1, A_{\text{context}}), \mathcal{A}_{\text{TermCons}}(\gamma_2, A_{\text{context}}))$
6: **end if**
7: **for each** $(v, \tau) \in \{(a, b) \mid b \in \{\gamma_1, \gamma_2\} \wedge a \in \mathcal{V}(b) \setminus \mathcal{V}(A_{\mathbb{S}}) \wedge \varphi \equiv \gamma_1 \star \gamma_2\}$ **do**
8: $A_{\mathbb{S}} \leftarrow A_{\mathbb{S}}[\mathcal{V}(\varphi)] \cap \mathcal{A}_{\text{TermProp}}(v, \tau, A_{\mathbb{S}}[v_\tau], A_{\text{context}})[\mathcal{V}(\varphi)]$
9: **end for**
10: **return** $A_{\mathbb{S}}$
11: **end function**

Before going into the details of the algorithm, let us define the relational automata construction semantics for the string predicate operations (Table 5.1). Encoding semantics of the string predicate operations in multi-track automata allows us to keep relations among the variables that appear in the predicate.

Let us consider the atomic string constraint $\varphi_{\mathbb{S}} \equiv \text{charat}(v, i) = \texttt{"a"}$ from Example 5.1. First, an automaton A_1 where $\mathcal{L}(A_1) = \vec{\Sigma}$ is constructed for the string term $\gamma_1 \equiv \text{charat}(v, i)$ and an automaton A_2 where $\mathcal{L}(A_2) = \{\texttt{"a"}\}$ is constructed for the string term $\gamma_2 \equiv \texttt{"a"}$ at line 3 using the term automata constructor function $\mathcal{A}_{\text{TermCons}}$ presented in Algorithm 9. Next, function $\mathcal{A}_{\mathbb{S}}$ calls an automaton construction function based on the string predicate. In our example, the string predicate $\star$ is an equality ($=$) constraint on string terms. Function $\mathcal{A}_{\mathbb{S}}$ calls $\mathcal{A}_{=}$ at line 3 to construct an automaton $A_{\mathbb{S}}$ given the automata A_1 and A_2 constructed for the terms γ_1 and γ_2. We provide the

Operation	**Automata Construction Semantics**
$\mathcal{A}_{=}(A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A) = \{(s_1, s_2) \mid s_1 \in \mathcal{L}(A_{\gamma_1}) \wedge s_2 \in \mathcal{L}(A_{\gamma_2}) \wedge s_1 = s_2\}$
$\mathcal{A}_{\neq}(A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A) = \{(s_1, s_2) \mid s_1 \in \mathcal{L}(A_{\gamma_1}) \wedge s_2 \in \mathcal{L}(A_{\gamma_2}) \wedge s_1 \neq s_2\}$
$\mathcal{A}_{<}(A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A) = \{(s_1, s_2) \mid s_1 \in \mathcal{L}(A_{\gamma_1}) \wedge s_2 \in \mathcal{L}(A_{\gamma_2}) \wedge s_1 < s_2\}$
$\mathcal{A}_{\leq}(A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A) = \{(s_1, s_2) \mid s_1 \in \mathcal{L}(A_{\gamma_1}) \wedge s_2 \in \mathcal{L}(A_{\gamma_2}) \wedge s_1 \leq s_2\}$
$\mathcal{A}_{>}(A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A) = \{(s_1, s_2) \mid s_1 \in \mathcal{L}(A_{\gamma_1}) \wedge s_2 \in \mathcal{L}(A_{\gamma_2}) \wedge s_1 > s_2\}$
$\mathcal{A}_{\geq}(A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A) = \{(s_1, s_2) \mid s_1 \in \mathcal{L}(A_{\gamma_1}) \wedge s_2 \in \mathcal{L}(A_{\gamma_2}) \wedge s_1 \geq s_2\}$
$\mathcal{A}_{\text{match}}(A_{\gamma}, A_{\rho})$	$\mathcal{L}(A) = \{s \mid s \in \mathcal{L}(A_{\gamma}) \wedge s \in \mathcal{L}(A_{\rho})\}$
$\mathcal{A}_{\overline{\text{match}}}(A_{\gamma}, A_{\rho})$	$\mathcal{L}(A) = \{s \mid s \in \mathcal{L}(A_{\gamma}) \wedge s \notin \mathcal{L}(A_{\rho})\}$
$\mathcal{A}_{\text{begins}}(A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A) = \{(s_1, s_2) \mid s_1 \in \mathcal{L}(A_{\gamma_1}) \wedge s_2 \in \mathcal{L}(A_{\gamma_2}) \wedge \exists t \in \Sigma^* \colon s_1 = s_2 t\}$
$\mathcal{A}_{\overline{\text{begins}}}(A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A) = \{(s_1, s_2) \mid s_1 \in \mathcal{L}(A_{\gamma_1}) \wedge s_2 \in \mathcal{L}(A_{\gamma_2}) \wedge \nexists t \in \Sigma^* \colon s_1 = s_2 t\}$
$\mathcal{A}_{\text{ends}}(A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A) = \{(s_1, s_2) \mid s_1 \in \mathcal{L}(A_{\gamma_1}) \wedge s_2 \in \mathcal{L}(A_{\gamma_2}) \wedge s_1 \in \Sigma^* \mathcal{L}(A_{\gamma_2}) \wedge s_2 \in \mathcal{L}(\text{suffixes}(A_{\gamma_1}))\}$
$\mathcal{A}_{\overline{\text{ends}}}(A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A) = \{(s_1, s_2) \mid \exists t_1 \in \mathcal{L}(A_{\gamma_1}), t_2 \in \mathcal{L}(A_{\gamma_2}) \colon s_1 \in \mathcal{L}(A_{\gamma_1}) \wedge s_2 \in \mathcal{L}(A_{\gamma_2}) \wedge s_1 \notin \Sigma^* t_2 \wedge s_2 \notin \mathcal{L}(\text{suffixes}(t_1))\}$
$\mathcal{A}_{\text{contains}}(A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A) = \{(s_1, s_2) \mid s_1 \in \mathcal{L}(A_{\gamma_1}) \wedge s_2 \in \mathcal{L}(A_{\gamma_2}) \wedge s_1 \in \Sigma^* \mathcal{L}(A_{\gamma_2}) \Sigma^* \wedge s_2 \in \mathcal{L}(\text{suffixes}(\text{prefixes}(A_{\gamma_1})))\}$
$\mathcal{A}_{\overline{\text{contains}}}(A_{\gamma_1}, A_{\gamma_2})$	$\mathcal{L}(A) = \{(s_1, s_2) \mid \exists t_1 \in \mathcal{L}(A_{\gamma_1}), t_2 \in \mathcal{L}(A_{\gamma_2}) \colon s_1 \in \mathcal{L}(A_{\gamma_1}) \wedge s_2 \in \mathcal{L}(A_{\gamma_2}) \wedge s_1 \notin \Sigma^* t_2 \Sigma^* \wedge s_2 \notin \mathcal{L}(\text{suffixes}(\text{prefixes}(t_1)))\}$

Table 5.1: Relational automata constructions semantics for $\mathcal{A}_{\text{sop}}$ and $\mathcal{A}_{\overline{\text{sop}}}$.

details of the automata constructions for string predicate operations in Table 5.1. The resulting automaton $A_{\mathbb{S}}$ where $\mathcal{L}(A) = \{\texttt{"a"}\}$ represents the values of the terms that satisfies the predicate. Automaton $A_{\mathbb{S}}$ includes tracks for the terms that are not constant; term $\gamma_1 \equiv \text{charat}(v, i)$ is represented with an auxiliary variable v_{γ_1}, and term γ_2 does not appear in the tracks since it represents a constant value. Line 7 iterates over the terms and collects the variables that appear in the term, but not in the automaton constructed so far. In our example, since $\mathcal{V}(A_{\mathbb{S}}) = \{v_{\gamma_1}\}$ and $\mathcal{V}(\gamma_1) = \{v, i, v_{\gamma_1}\}$, for loop at line 7 collects tuples (v, γ_1) and (i, γ_1) and executes the function $\mathcal{A}_{\text{TermProp}}$ to propagate the result of the string predicate operation back to the variables. Executions of the line 8 adds tracks for the variables v and i to the automaton $A_{\mathbb{S}}$. Finally, automaton $A_{\mathbb{S}}$ where $\mathcal{V}(A_{\mathbb{S}}) = \mathcal{V}(\varphi_{\mathbb{S}})$ is returned.

We now discuss the functions $\mathcal{A}_{\text{TermCons}}$ and $\mathcal{A}_{\text{TermProp}}$ that are used during string automata construction. Given a term τ and A_{context}, the function $\mathcal{A}_{\text{TermCons}}$ generates an automaton by recursive decomposition into subformulas (Algorithm 9). If τ is a regular expression term ρ, an automaton A_ρ is constructed using standard regular expression DFA constructions extended to multi-track DFA. If τ is an integer constant, A_n is constructed to recognize all strings of length n. For string variable terms, the current set of values for the variable is extracted from A_{context} using π.

Integer variable terms are encoded as binary strings in our automata representation. One of our contributions in this dissertation is to provide a function ToStrEncoding that converts binary encoded set of values of an integer variable into a unary encoded set of values. Function ToStrEncoding enables us to compute an automaton for the mixed constraints and we discuss the details of the function in Section 5.3.

If τ corresponds to a term operation ($\odot \in \{\text{length}, \ldots, \text{replaceall}\}$), the corresponding term automata construction function $\mathcal{A}_\odot$ is called. Term automata constructor functions are discussed in Chapter 4.3.2.

Algorithm 9 Term automata constructor

Input: Term τ : target term, Automaton A : values of variables where $\mathcal{V}(A) = \mathcal{V}(\tau)$
Output: Automaton A : constructed as a result of term τ

1: **function** $\mathcal{A}_{\text{TermCons}}(\tau, A_{\text{context}})$
2: **if** $\tau \equiv \rho$ **then**
3: **return** A_ρ where $\mathcal{L}(A_\rho) = \{s \mid s \in \mathcal{L}(\rho)\}$.
4: **else if** $\tau \equiv n$ **then**
5: **return** A_n where $\mathcal{L}(A_n) = \{s \mid |s| = n\}$.
6: **else if** $\tau \equiv v_s$ **then**
7: **return** $A_{\text{context}}[\{v_s\}]$
8: **else if** $\tau \equiv v_i$ **then**
9: **return** ToStrEncoding$(A_{\text{context}}[\{v_i\}])$
10: **else if** $\tau \equiv \odot(\tau_1, \ldots, \tau_n)$ **then** ▷ $\odot \in \{\text{length}, \ldots, \text{replaceall}\}$.
11: **return** $\mathcal{A}_\odot(0, \mathcal{A}_{\text{TermCons}}(\tau_1, A_{\text{context}}), \ldots, \mathcal{A}_{\text{TermCons}}(\tau_n, A_{\text{context}}))$
12: **end if**
13: **end function**

Let us consider the running example again. For the string term $\gamma_1 \equiv \text{charat}(v, i)$, function $\mathcal{A}_{\text{TermCons}}$ first generates an automaton A_v for the variable term v using the automaton that accepts strings represented by the variable (line 7). Next, it generates an automaton A_i for the variable term i at line 9. Next, for the term $\gamma_1 \equiv \text{charat}(v, i)$, $\mathcal{A}_{\text{charat}}$ (where $\odot = \text{charat}$) constructs an automaton A_1 given the automata A_v and A_i as parameters at line 11. Initially variables v and i are unconstrained, hence the resulting automaton A_1 accepts all characters. Similarly, $\mathcal{A}_{\text{TermCons}}$ generates an automaton A_2 where $\mathcal{L}(A_2) = \{\texttt{"a"}\}$ for the string term $\gamma_2 \equiv \texttt{"a"}$.

Now, we discuss the details of the function $\mathcal{A}_{\text{TermProp}}$. Given a variable v, a term τ that contains the variable v, an automaton A_τ that characterizes the values of the term τ, and A_{context}, function $\mathcal{A}_{\text{TermProp}}$ recursively computes the values of the variable v that results in the values accepted by the automaton A_τ when the term operation that corresponds to τ is applied. When a term τ corresponds to a string variable, the updated set of values for the variable is computed by intersecting the propagated automaton A_τ with A_{context}. Since values of integer variables are encoded as binary strings, we need to convert the unary encoded set of values into binary encoded set of values using the function ToBinEncoding. We discuss the details of the function ToBinEncoding in Section 5.3.

If a term τ corresponds to a term operation ($\odot \in \{\text{length}, \ldots, \text{replaceall}\}$), the corresponding term automata propagator function $\mathcal{A}_{\odot}$ is called to construct an automaton for the sub terms that contain the variable v given the automata for the term τ and other sub terms. Term automata constructor functions are discussed in Chapter 4.3.2.

Algorithm 10 Term automata propagator

Input: Variable v: target variable, Term τ : current term, Automaton A_τ : result of term τ, Automaton A : values of variables where $\mathcal{V}(A) = \mathcal{V}(\tau)$
Output: Automaton A_v : solutions to variable v

```
1: function A_TermProp(v, τ, A_τ, A)
2:     if τ ≡ v_s then
3:         return A_τ ∩ A[{v_s}]
4:     else if τ ≡ v_i then
5:         return ToBinEncoding(A_τ) ∩ A[{v_i}]
6:     else if τ ≡ ⊙(τ_1, ..., τ_n) then                ▷ ⊙ ∈ {length, ..., replaceall}.
7:         for all i ∈ {k | 1 ≤ k ≤ n ∧ v ∈ V(τ_k)} do
8:             A_τi ← A_⊙(i, A_τ, A_TermCons(τ_1, A), ..., A_TermCons(τ_n, A))
9:             A ← A ∩ A_TermProp(v, τ_i, A_τi, A)[V(A)]
10:        end for
11:        return π(A, {v})
12:    end if
13: end function
```

Let us consider the running example again for discussion of the Algorithm 8. Function $\mathcal{A}_{\text{TermProp}}$ is called twice at line 8, once for string variable v ($\mathcal{A}_{\text{TermProp}}(v, \gamma_1, \pi(A_{\mathbb{S}}, v_{\gamma_1}))$) and once for integer variable i ($\mathcal{A}_{\text{TermProp}}(i, \gamma_1, \pi(A_{\mathbb{S}}, v_{\gamma_1}))$). For the string variable v and the term $\gamma_1 \equiv \text{charat}(v, i)$, $\mathcal{A}_{\text{charat}}$ (where $\odot = \text{charat}$) constructs an automaton A_v where $\mathcal{L}(A_v) = \{s \mid s = \exists s_1, s_2 \in \Sigma^* \colon s = s_1 \mathtt{a} s_2\}$. Informally, automaton that corresponds to the charat term accepts only character a. Since integer variable i is unconstrained, we only say that variable v must contain character a. Note that, this is an over-approximation. Next, the automaton A_v is intersected with the automaton that represents the initial values of the variable v. Similarly, for the integer variable i, $\mathcal{A}_{\text{charat}}$ constructs an automaton A_i where $\mathcal{L}(A_i) = \{n \mid n \geq 0\}$ since character indexes in a string start from 0. In the last step, A_i is converted into an automaton that accepts binary strings for the integers and intersected with the initial values of the variable i.

To sum up, function $\mathcal{A}_{\text{TermProp}}$ restricts the values of the variables v and i based on the result of the charat operation.

5.2 Integer Constraint Solving

Integer automata constructor $\mathcal{A}_{\mathbb{Z}}$ (Algorithm 11) handles arithmetic formulae $\varphi_{\mathbb{Z}}$ consisting of linear equalities ($=$), disequalities ($\neq$), and inequalities ($<, \leq, >, \geq$).

Given an atomic numeric constraints $\varphi_{\mathbb{Z}}$, function $\mathcal{A}_{\mathbb{Z}}$ first extracts the coefficients of the of the integer terms in the form $\sum_{i=1}^{n} c_i \cdot \beta_i + c_0 \otimes 0$ where c_i denotes integer coefficients and β_i denotes atomic integer terms and $\otimes \in \{=, \neq, >, \geq, \leq, <\}$. Then, the automata construction techniques that rely on a binary adder state machine construction is used to construct an automaton for the arithmetic constraint [44]. An integer term may correspond to an operation between strings and integers, such as $x = \text{length}(v) + 1$. In such cases, an auxiliary integer variable, say $l_s = v$, is used to track the values of the result of the term operation corresponding to satisfying assignments of the variables appearing in the term. The loop condition at line 4 checks if there is any term operation involved in the integer constrain. When the loop evaluates to true, the generated integer automaton is first refined further and then extended with the variables that appear in the term. Note that, if the term operation contains a string variable, $\varphi_{\mathbb{Z}}$ is a mixed integer constraint.

Let us consider the Example 5.2 again. It contains an atomic integer constraint $\varphi_{\mathbb{Z}} \equiv x = \text{length}(v) + 1$. Function CollectCoeffs in Algorithm 11 extracts the coefficients $c_0 = -1$, $c_1 = 1$, $c_2 = -1$ for the integer terms $\beta_1 \equiv x$, $\beta_2 \equiv \text{length}(v)$ where $\otimes$ is equality ($=$). An automaton $A_{\mathbb{Z}}$ is constructed where $\mathcal{V}(A_{\mathbb{Z}}) = \{x, v_l\}$ and v_l is an auxiliary variable used for the length term. An example accepting tuple for the automaton is $(-1, -2)$. Next, the for loop at line 4 identifies the tuple $(v, \text{length}(v))$. First, $\mathcal{A}_{\text{TermCons}}$

Algorithm 11 Relational integer automata constructor

Input: Formula φ: an atomic constraint, Automaton A_{context} : initial values of the variables where $\mathcal{V}(A) = \mathcal{V}(\varphi)$
Output: Automaton $A_{\mathbb{Z}}$: updated values of the variables

1: **function** $\mathcal{A}_{\mathbb{Z}}(\varphi, A_{\text{context}})$
2: $\quad \otimes, c_0, c_1, \beta_1, \ldots, c_n, \beta_n \leftarrow \text{CollectCoeffs}(\varphi)$
3: $\quad A_{\mathbb{Z}} \leftarrow \mathcal{A}_{\otimes}(c_0, c_1, \beta_1, \ldots, c_n, \beta_n)$ ▷ construction based on the techniques in [44].
4: $\quad$ **for all** $(v, \tau) \in \{(a, b) \mid b \in \{\beta_1, \ldots, \beta_n\} \wedge a \in \mathcal{V}(b) \setminus \mathcal{V}(A_{\mathbb{Z}})\}$ **do**
5: $\quad\quad A_{\mathbb{Z}} \leftarrow A_{\mathbb{Z}} \cap \text{ToBinEncoding}(\mathcal{A}_{\text{TermCons}}(\tau, A_{\text{context}}))[\mathcal{V}(A_{\mathbb{Z}})]$
6: $\quad\quad A_{\mathbb{Z}} \leftarrow A_{\mathbb{Z}}[\mathcal{V}(\varphi)] \cap \mathcal{A}_{\text{TermProp}}(v, \tau, \text{ToStrEncoding}(A_{\mathbb{Z}}[v_\tau], A_{\text{context}})[\mathcal{V}(\varphi)]$
7: $\quad$ **end for**
8: $\quad$ **return** $A_{\mathbb{Z}}$
9: **end function**

is called to restrict the integer constraint further based on the length of the variable v. Since variable v can represent strings with any length, it restricts the variable as $v_l \geq 0$. At this point, an accepting tuple for the automaton $A_{\mathbb{Z}}$ is $(2, 1)$. Next, $\mathcal{A}_{\text{TermCons}}$ is called to propagate back the values of the v_l to the string variable v. After that, the automaton $A_{\mathbb{Z}}$ is extended with a track for the string variable v. An example accepting tuple for the final automaton is $(2, 1, \texttt{"a"})$. Note that, functions ToBinEncoding and ToStrEncoding are used for conversion between different encodings in order to represent the relation between string and integer variables. In the next section, we discuss the details of both of these functions.

5.3 Mixed Constraint Solving

During automata construction, the ToStrEncoding function is used to convert binary integer automata to length automata to refine string solutions, and the ToBinEncoding function is used to convert length automata to binary integer automata to refine integer solutions. Note that, both functions are always called on an automaton that is projected onto a single track in the algorithms discussed in the previous subsections.

A linear set L_i is given by $\{a_i + b_i k : k \in \mathbb{Z}\}$, for some $a_i, b_i \in \mathbb{Z}$. A semilinear set S is

a finite union of linear sets $S = \cup_i L_i$. For any single-track binary integer DFA A, $\mathcal{L}(A)$ forms a semilinear set S [4]. Given a binary integer DFA A constructed from linear integer arithmetic constraints, we wish to recover the semilinear set that it represents. We adopt algorithms proposed in [4] but further propose BinToSemSet function (Algorithm 12) to construct semilinear sets of binary integer automata (instead of approximating them as in [4]).

Algorithm 12 Semilinear set extraction from a binary encoded integer automaton

Input: Automaton A: a binary encoded integer automaton, Number i : bit bound
Output: Semilinear set S : union of linear sets

```
 1: function BinToSemSet(A, i)
 2:     if L(A) is a finite set then
 3:         return S where S = {n | n ∈ L(A)}
 4:     else if i > 2 × |A| then
 5:         return S where S = {n | (n < 2^i ∧ n ∈ L(A)) ∨ n > 2^i}
 6:     else
 7:         N ← GetValues(A, i)
 8:         while N ≠ ∅ do
 9:             a ← RemoveMin(N), N' ← N
10:             while N' ≠ ∅ do
11:                 b ← RemoveMin(N')
12:                 construct S where S = {n | n = a + (b − a) × k ∧ k ≥ 0}
13:                 if S ⊆ L(A) then
14:                     return S ∪ BinToSemSet(A \ A(S), 1)
15:                 end if
16:             end while
17:         end while
18:         return BinToSemSet(A, i + 1)
19:     end if
20: end function
```

Given an automaton A and a bit-width bound i on recursion (initially $i = 1$), BinToSemSet(A, i) recursively constructs a semilinear set S, s.t., $\mathcal{L}(A) = S$ if $\mathcal{L}(A)$ is a semilinear set; $\mathcal{L}(A) \subseteq S$, otherwise. At recursive steps, once a linear set S is found from a given automaton A, we add the set S to the result of the next recursive call where we pass a new automaton $A \setminus \mathcal{A}(S)$ such that $\mathcal{L}(A \setminus \mathcal{A}(S)) = \mathcal{L}(A) \setminus S$ and the new bit-width bound is reset to 1. In that recursive step, the algorithm tries to find a linear set that forms from a minimal pair of accepting values in $\mathcal{L}(A)$.

The procedure conducts an exhaustive search by enumerating all potential pairs in the set of accepting values that have their bit-width bounded by i. GetValues(A, i) returns the set $\{n \mid n < 2^i \wedge n \in \mathcal{L}(A)\}$. If we cannot find a linear set given the bit-width bound i, we increase i by one and recurse the procedure. If the recursion ends at line 3, we return a semilinear set S where $S = \mathcal{L}(A)$. If the recursion returns because that bit-width bound is greater than a threshold, we return a semilinear set that over approximates $\mathcal{L}(A)$. The threshold is set as $2 \times |A|$ to ensure that before the termination at least two numbers of any linear set in $\mathcal{L}(A)$ have been checked.

Example Let us consider the following example:

$$i = 2 \times j \wedge \text{length}(v) = i \tag{5.3}$$

Example 5.3 is a conjunction of an atomic integer constraints $\varphi_1 \equiv i = 2 \times j$ and $\varphi_2 \equiv \text{length}(v) = i$. The constraint φ_2 is also a mixed constraint as it contains both a string and an integer variable. The automaton constructed for the atomic constraint φ_1 $(A_{(i,j)})$ is shown in Figure 5.1. That construction happens with the call to Algorithm 11 at line 7 in Algorithm 6. Atomic constraint φ_2 is a mixed constraint, Figure 5.1 shows the automata $(A_{(v_l,i)}, A_v)$ constructed for the it. The automata constructed for φ_2 refers to the constructions that happened before handling the mixed constraints at line 4 in Algorithm 11. To better explain the algorithm visually, we will keep string and integer variables in separate automata.

Let v_l be an auxiliary variable that represents bitwise encodings of the lengths of the strings that are represented with the variable v. As shown in Figure 5.1, the automaton $A_{(v_l,i)}$ accepts negative integers however, string lengths cannot be negative integers. In order to refine the integer automaton based on the string automaton we need to handle the mixed constraint. Let v_u be an auxiliary variable that represents the unary encod-

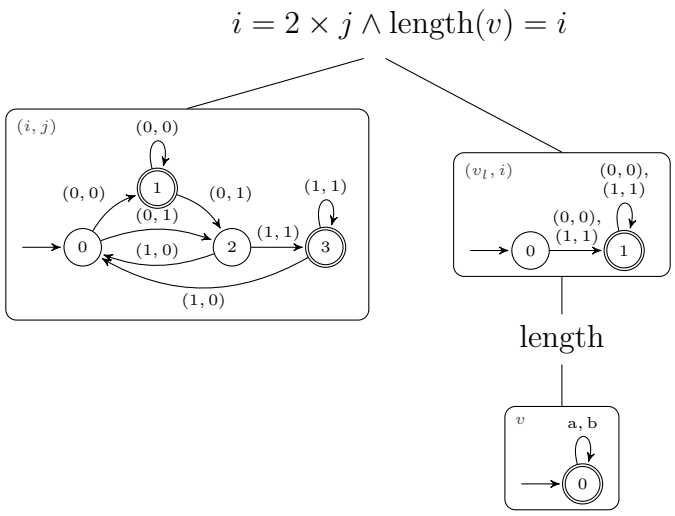

Figure 5.1: Automata constructed for the atomic constraints $\varphi_1 \equiv i = 2 \times j$ and $\varphi_2 \equiv \text{length}(v) = i$ before handling the mixed constraint.

ing of the lengths of the the strings that are represented with the variable v. A unary encoding can be obtained by mapping string alphabet Σ to a symbol λ where $\lambda \notin \Sigma$. Figure 5.2 shows the details of conversions between string and integer automata while handling mixed constraints. Figure 5.2a shows the steps of the string to integer automaton conversion which is called at line 5 in Algorithm 11. Figure 5.2b shows the steps of the integer to string automaton conversion which is called at line 6 in Algorithm 11.

Figure 5.3a shows the automata constructed after handling the mixed constraint φ_2. After constructing automata for the atomic constraints φ_1 and φ_2, the resulting automata are conjuncted as presented in Algorithm 6. Figure 5.3b shows the result of the conjunction before applying the refine function at line 11 in Algorithm 6. Note that, the resulting integer automaton now accepts positive even numbers for the variables v_l and i due to atomic constraint φ_1. However, string automaton for the variable v still accepts any string.

The result of a mixed constraint is over-approximated as a multi-track automaton cannot encode the relation between an unbounded string variable and an unbounded in-

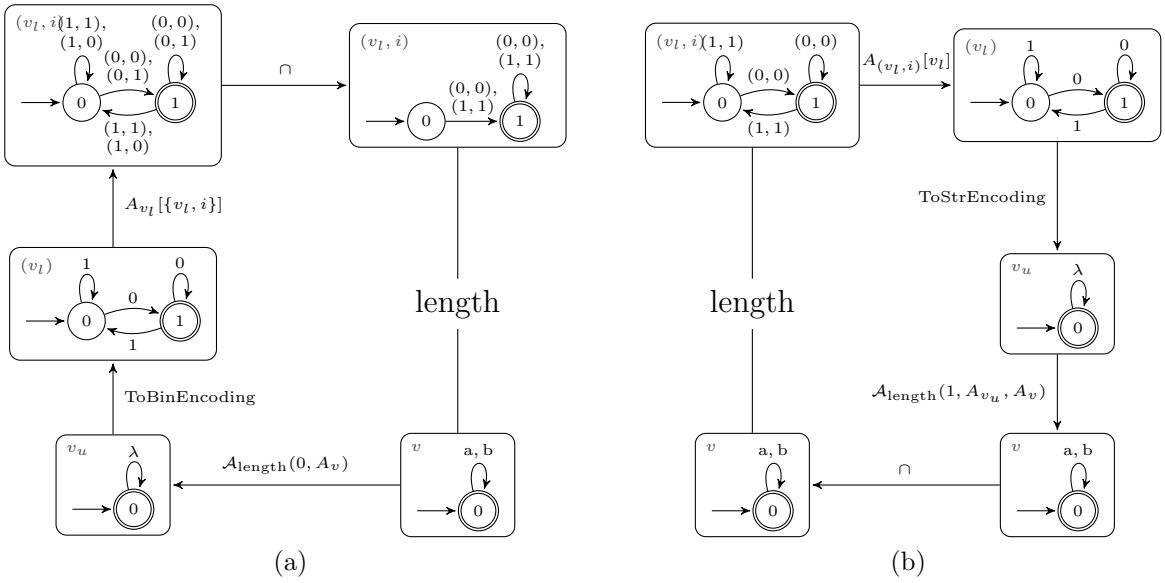

Figure 5.2: Handling mixed constraint $\varphi_2 \equiv \text{length}(v) = i$: (a) string to integer automaton conversion , and (b) integer to string automaton conversion.

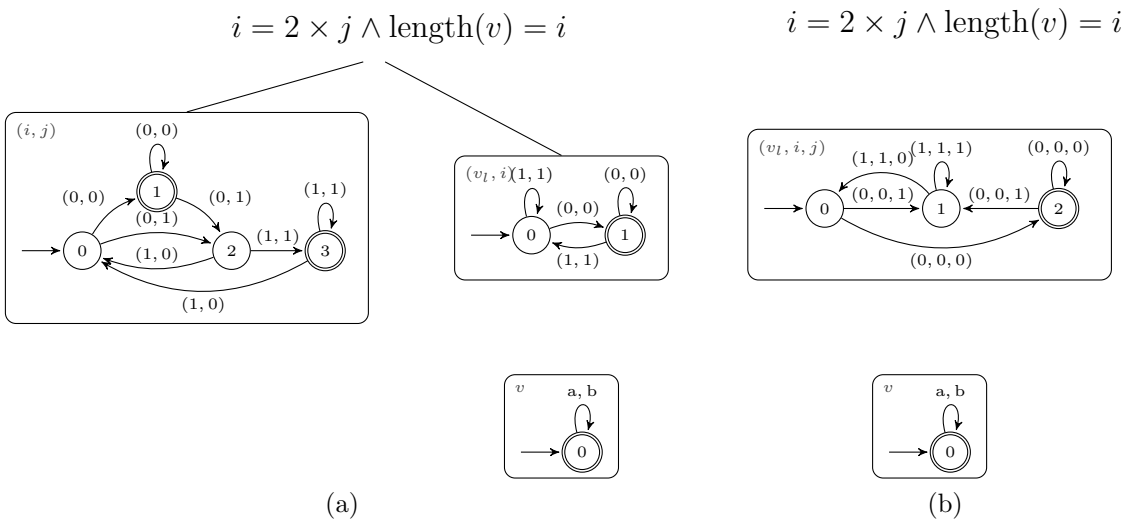

Figure 5.3: (a) result of the mixed constraint handling step , and (b) result of the conjunction without applying Refine step.

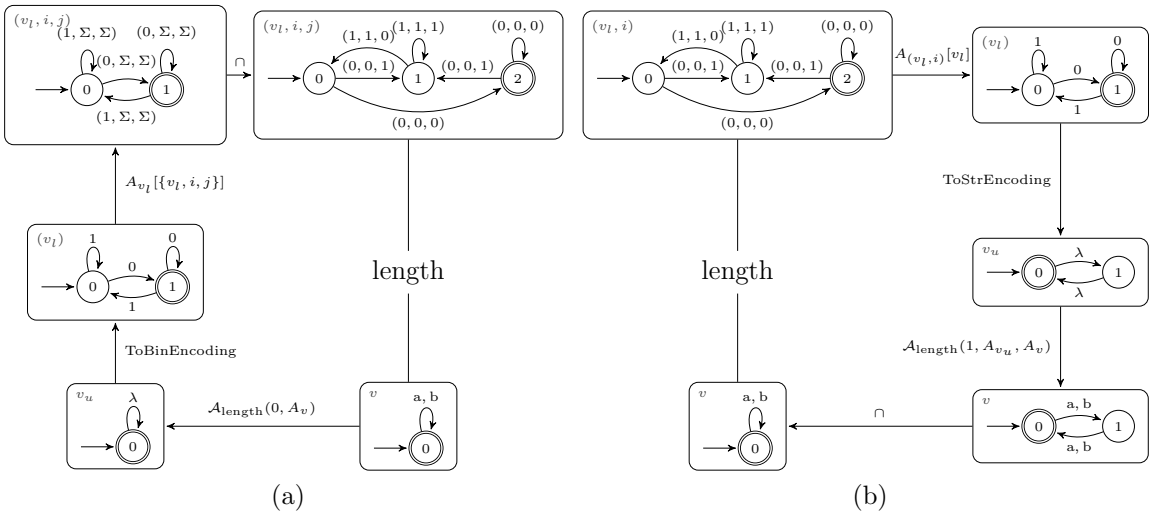

Figure 5.4: Handling mixed constraint $\varphi_2 \equiv \text{length}(v) = i$ inside Refine: (a) string to integer automaton conversion , and (b) integer to string automaton conversion.

teger variable (in bitwise encoding). Refine function (Algorithm 7) takes the constructed automata in Figure 5.3b as input and generates refined automata by re solving the mixed constraint φ_2 again. Figure 5.4a and Figure 5.4b shows string to integer automaton and integer to string automaton steps, respectively. String automaton for the variable v does not change after handling the conjunction. As a result of that, string to integer conversion does not result in any changes in the integer automaton. However, integer to string conversion updates the string automaton based on the update in integer automaton.

Figure **??**hows the final automata constructed for the input formula $\varphi \equiv i = 2 \times j \wedge \text{length}(v) = i$.

The example formula $i = 2 \times j \wedge \text{length}(v) = i$ contains a mixed constraint where an atomic integer constraint involves a string and an integer variable. Similarly, an atomic string constraint can involve an integer variable and a string variable. Consider the example $i = 2 \times j \wedge \text{charat}(v, i) =$ "a". Figure 5.6 shows automata constructed for the example $i = 2 \times j \wedge \text{charat}(v, i) =$ "a" where the mixed constraint $\text{charat}(v, i) =$ "a" handled using the conversion functions for the string and integer automata.

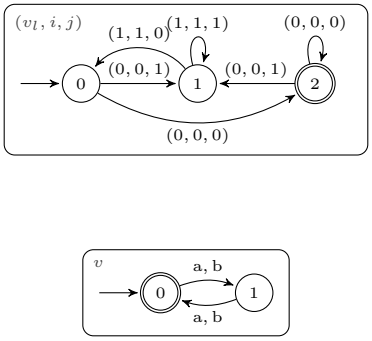

Figure 5.5: Automata constructed for the formula $\varphi \equiv i = 2 \times j \wedge \text{length}(v) = i$.

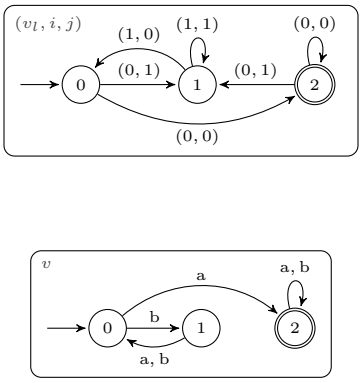

Figure 5.6: Automata constructed for the formula $\varphi \equiv i = 2 \times j \wedge \text{charat}(v, i) =$ "a".

Chapter 6

Constraint Simplification Heuristics

In this chapter we present a set of heuristics for improving both the precision and the efficiency of of our constraint solver.

Term Re-Write Rules: All terms are first reduced with respect to a re-write system based on a set of rules (Fig. 6.1). These rules include both term normalization rules and tautological simplifications of atomic constraints. Here, i, j are distinct integer constants, t, v are distinct string constants and $\gamma_1, \gamma_2, \gamma_3$ are (not necessarily distinct) string terms.

Dependency Analysis: To reduce the amount of work required to solve a constraint,

$$
\begin{array}{cccc}
\varphi \wedge \varphi \rightarrow \varphi & \varphi \vee \varphi \rightarrow \varphi & \varphi \vee \top \rightarrow \top & \\
\varphi \wedge \top \rightarrow \varphi & \varphi \vee \bot \rightarrow \varphi & \varphi \wedge \bot \rightarrow \bot & \\
 & & & |\epsilon| \rightarrow 0 \\
0 \times \beta \rightarrow 0 & \beta - 0 \rightarrow \beta & \beta = \beta \rightarrow \top & i \neq j \rightarrow \top \\
1 \times \beta \rightarrow \beta & -(-\beta) \rightarrow \beta & \beta \neq \beta \rightarrow \bot & |v_{s_1}.v_{s_2}| \rightarrow |v_{s_1}| + |v_{s_2}| \\
\beta + 0 \rightarrow \beta & \neg(\neg\beta) \rightarrow \beta & i = j \rightarrow \bot & \\
 & & & t.\gamma_1 = t.\gamma_2 \rightarrow \gamma_1 = \gamma_2 \\
\gamma.\epsilon \rightarrow \gamma & \gamma = \gamma \rightarrow \top & t.\gamma_1 \neq v.\gamma_2 \rightarrow \top & \gamma_1.t = \gamma_2.t \rightarrow \gamma_1 = \gamma_2 \\
\epsilon.\gamma \rightarrow \gamma & \gamma \neq \gamma \rightarrow \bot & t.\gamma_1 = v.\gamma_2 \rightarrow \bot & \text{ends}(\gamma_2.\gamma_1, \gamma_1) \rightarrow \top \\
t_1.t_2 \rightarrow t_1 t_2 & t = v \rightarrow \bot & \gamma_1.t \neq \gamma_2.v \rightarrow \top & \text{begins}(\gamma_1.\gamma_2, \gamma_1) \rightarrow \top \\
\gamma \in t \rightarrow \gamma = t & t \neq v \rightarrow \top & \gamma_1.t = \gamma_2.v \rightarrow \bot & \text{contains}(\gamma_2.\gamma_1.\gamma_3, \gamma_1) \rightarrow \top
\end{array}
$$

Figure 6.1: Term reduction rules.

we note that not all variables of a constraint need be counted together. We define the *constraint graph* of a formula φ to be the graph defined on the set of variables of φ where an edge exists between any two variables if they appear in the same clause of φ. This constraint graph can be decomposed into a finite set of connected components. A connected component C is a maximal subgraph such that if $u, v \in C$ then there exists a path between u and v in C.

Constraints on any given variable depend only on variables within its connected component. This allows us to decompose a formula based on connected components, solve and count each component individually, and then take the product of the results to obtain accurate counts for tuples of variables. This results in smaller automata and faster computation.

Equivalence Classes: The variables of a formula φ can be partitioned into equivalence classes so that any pair of given variables x, y are in the same equivalence class only if the have the same solution set. In our implementation, we construct these equivalence classes based on equality clauses. Every term, variable or otherwise, begins in its own equivalence class and for every equality clause, the equivalence classes of the left and right sides are merged.

From each equivalence class, we choose a representative. Priority in this choice is given to constant terms then to variables. Each variable in the equivalence class is then replaced by this representative in the formula φ. This optimization can result in the elimination of variables from φ, and hence tracks from its DFA, without any loss of precision in counting.

Implication Rules: As noted previously, our automata construction for some constraints can be imprecise. However, precision can be improved for some of these constraints by augmenting the original formula φ with clauses implied by φ. We present a set of implication rules which define the augmenting clauses added to φ in the presence

$$\gamma_1.\text{contains}(\gamma_2) \rightarrow |\gamma_1| \geq |\gamma_2| \qquad \neg\gamma.\text{contains}(t) \rightarrow \neg\gamma.\text{begins}(t) \qquad \neg\gamma.\text{ends}(t) \rightarrow \gamma \neq t$$
$$\gamma_1.\text{begins}(\gamma_2) \rightarrow |\gamma_1| \geq |\gamma_2| \qquad \gamma_1.\gamma_2 = \gamma_3.\gamma_4 \rightarrow \; |\gamma_1| + |\gamma_2| = |\gamma_3| + |\gamma_4|$$
$$\gamma_1.\text{ends}(\gamma_2) \rightarrow |\gamma_1| \geq |\gamma_2| \qquad \gamma_1.\gamma_2 = \gamma_3 \rightarrow \; |\gamma_1| + |\gamma_2| = |\gamma_3| \wedge \gamma_3.\text{begins}(\gamma_1)$$

Figure 6.2: Implication rules.

of certain imprecise constraints in Fig. 6.2.

Chapter 7

Automata-based Model Counting

In this chapter, we describe how to perform parameterized model counting by making use of the automata constructed by our constraint solving procedure. The *model counting problem* is to determine the size of $[\![\varphi]\!]$, which we denote $\#[\![\varphi]\!]$. A formula can have infinitely many models. However, we can count the number of models within an infinite space of solutions restricted to a finite range for the free variables. Hence, we perform *parameterized model counting* for string and integer constraints, in which $\#[\![\varphi]\!](b_{\mathbb{S}}, b_{\mathbb{Z}})$ is a function over parameters $b_{\mathbb{S}}$, which bounds the length of string solutions, and $b_{\mathbb{Z}}$, which bounds the bit-length representation of integer solutions.

The constraint solving procedure described in Chapter 4 produces a final automaton, A, for each variable in a given formula. The constraint solving procedure in Chapter 5 produces a final DFA, A, that contains multi-track solution sub-automata $A_{\mathbb{S}}$ and $A_{\mathbb{Z}}$. The model counting techniques we discuss here works for any DFA whether it is a single-track or multi-track. The only difference is in the interpretation of the count results; the former counts solutions to a single variable, whereas the latter counts solutions to tuples of variables. When counting for tuples, the separation of string and integer automata may lose some relational information between string and integer variables, but we can

multiply the model counts for each automaton in order to give a sound upper bound on the number of models for tuples of integer and string variables. We make use of two functions $\#F_{A_{\mathbb{S}}}(k)$ and $\#f_{A_{\mathbb{Z}}}(k)$ to count string and integer models respectively. The functions $\#F_{A_{\mathbb{S}}}(k)$ works identical for single-track ($\#F_A(k)$) and multi-track automata.

We rely on the observation that counting the number of strings of length k in a regular language, L, is equivalent to counting the number of accepting paths of length k in the DFA that accepts L. That is, by using a DFA representation, we reduce the parameterized model counting problem to counting the number of paths of a given length in a graph. In a DFA, there is exactly one accepting path for every recognized string. Thus, if we are interested in computing only string models or only integer models, there is no loss of precision due to the the model counting procedure; any loss of precision for strings comes from the over-approximations of non-regular constraints in the solving phase, and for pure integer constraints, the model counting procedure is precise because relational integer solution automata construction is precise.

We employ algebraic graph theory [45] and analytic combinatorics [46] to perform model counting. In our method, model counting corresponds exactly to counting the accepting paths of the constraint DFA up to a given length bound k. This problem can be solved using dynamic programming techniques in $O(k \cdot |\delta|)$ time where δ is the DFA transition relation [47, 48]. However, for each different bound, the dynamic programming technique requires another traversal of the DFA graph.

A preferable solution is to derive a symbolic function that given a length bound k outputs the number of solutions within bound k. One way to achieve this is to use the *transfer matrix method* [49, 46, 50] to produce an ordinary generating function which in turn yields a linear recurrence relation that is used to count constraint solutions. The transfer matrix T of A is a matrix where $T_{i,j}$ is the number of transitions from state i to state j. We will briefly review the necessary background and then describe the model

counting algorithm.

Given a DFA A, consider its corresponding language L. Let $L_i = \{w \in L : |w| = i\}$, the language of strings in L with length i. Then $L = \bigcup_{i \geq 0} L_i$. Define $|L_i|$ to be the cardinality of L_i. The cardinality of L can be computed by the sum of a series $a_0, a_1, \ldots, a_i, \ldots$ where each a_i is the cardinality of the corresponding language L_i, i.e., $a_i = |L_i|$.

For example, recall the final DFA A for the formula $\varphi \equiv \neg\text{match}(x, (01)^*) \wedge \text{length}(x) \geq 1$ in Figure 4.2. Let $\mathcal{L}(A)$ ($\mathcal{L}(A) = [\![\varphi, x]\!] = [\![\varphi]\!]$) be the language over $\Sigma = \{0, 1\}$ that satisfies the formula φ. Then $\mathcal{L}(A)$ is described by the expression $\Sigma^* - (01)^*$. In the language $\mathcal{L}(A)$, we have zero strings of length 0 ($\varepsilon \notin \mathcal{L}(A)$), two strings of length 1 ($\{0, 1\}$), three strings of length 3 ($\{00, 10, 11\}$), and so on. The sequence is then $a_0 = 0, a_1 = 2, a_2 = 3, a_3 = 8, a_4 = 15$, etc. For any length i, $|\mathcal{L}(A)_i|$, is given by a 3^{rd} order linear recurrence relation:

$$\begin{aligned} &a_0 = 0, a_1 = 2, a_2 = 3 \\ &a_i = 2a_{i-1} + a_{i-2} - 2a_{i-3} \quad \text{for } i \geq 3 \end{aligned} \tag{7.1}$$

In fact, using standard techniques for solving linear homogeneous recurrences, we can derive a closed form solution to determine that

$$|\mathcal{L}(A)_i| = (1/2)(2^{i+1} + (-1)^{i+1} - 1). \tag{7.2}$$

In the following discussion we give a general method based on generating functions for deriving a recurrence relation and closed form solution that we can use for model counting.

Generating Functions: Given the representation of the size of a language L as a sequence $\{a_i\}$ we can encode each $|L_i|$ as the coefficients of a polynomial, an ordinary generating

function (GF). The *ordinary generating function* of the sequence $a_0, a_1, \ldots, a_i, \ldots$ is the infinite polynomial [49, 46]

$$g(z) = \sum_{i \geq 0} a_i z^i \tag{7.3}$$

Although $g(z)$ is an infinite polynomial, $g(z)$ can be interpreted as the Taylor series of a finite rational expression. I.e., we can also write $g(z) = p(z)/q(z)$, where $p(z)$ and $q(z)$ are finite degree polynomials. If $g(z)$ is given as a finite rational expression, each a_i can be computed from the Taylor expansion of $g(z)$:

$$a_i = \frac{g^{(i)}(0)}{i!} \tag{7.4}$$

where $g^{(i)}(z)$ is the i^{th} derivative of $g(z)$. We write $[z^i]g(z)$ for the i^{th} Taylor series coefficient of $g(z)$. Returning to our example, we can write the generating function for $|\mathrm{L}_i^x|$ both as a rational function and as an infinite Taylor series polynomial. The reader can verify the following equivalence by computing the right hand side coefficients via equation (7.4).

$$g(z) = \frac{2z - z^2}{1 - 2z - z^2 + 2z^3} = 0z^0 + 2z^1 + 3z^2 + 8z^3 + 15z^4 + \ldots \tag{7.5}$$

Generating Function for a DFA: Given a DFA A and length k we can compute the generating function $g_A(z)$ such that the k^{th} Taylor series coefficient of $g_A(z)$ is equal to $|\mathrm{L}_k(A)|$ using the transfer-matrix method [49, 46].

We first apply a transformation and add an extra state, s_{n+1}. The resulting automaton is a DFA A' with λ-transitions from each of the accepting states of A to s_{n+1} where λ is a new padding symbol that is not in the alphabet of A. Thus, $\mathcal{L}(A') = \mathcal{L}(A) \cdot \lambda$ and furthermore $|\mathcal{L}(A)_i| = |\mathcal{L}(A')_{i+1}|$. That is, the augmented DFA A' preserves both the

language and count information of A. Recalling the final automaton from Figure 4.2, the corresponding augmented DFA is shown in Figure 7.1b. (Ignore the dashed λ transition for the time being.)

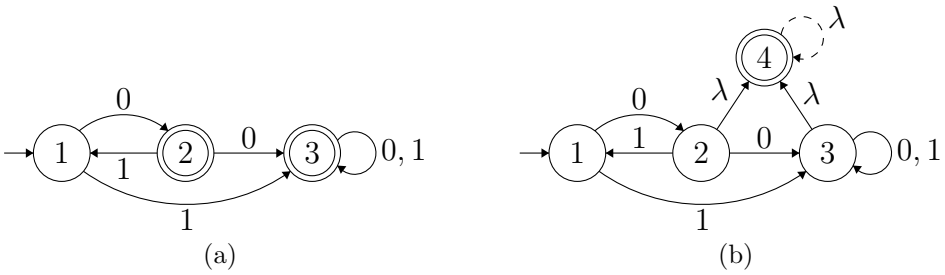

Figure 7.1: (a) The original DFA A, and (b) the augmented DFA A' used for model counting (sink state omitted).

From A' we construct the $(n+1) \times (n+1)$ transfer matrix T. A' has $n+1$ states $s_1, s_2, \ldots s_{n+1}$. The matrix entry $T_{i,j}$ is the number of transitions from state s_i to state s_j. Then the generating function for A is

$$g_A(z) = (-1)^n \frac{\det(I - zT : n+1, 1)}{z \det(I - zT)}, \tag{7.6}$$

where $(M : i, j)$ denotes the matrix obtained by removing the i^{th} row and j^{th} column from M, I is the identity matrix, $\det M$ is the matrix determinant, and n is the number of states in the original DFA A. The number of accepting paths of A with length exactly k, i.e. $|\mathcal{L}(A)_k|$, is then given by $[z^k]g_A(z)$ which can be computed through symbolic differentiation via equation 7.4.

For our running example, we show the transition matrix T and the terms $(I - zT)$ and $(I - zT : n, 1)$. Here, $T_{1,2}$ is 1 because there is a single transition from state 1 to state 2, $T_{3,3}$ is 2 because there are two transitions from state 3 to itself, $T_{2,4}$ is 1 because there is a single (λ) transition from state 2 to state 4, and so on for the remaining entries.

$$T = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}, I - zT = \begin{bmatrix} 1 & -z & -z & 0 \\ -z & 1 & -z & -z \\ 0 & 0 & 1-2z & -z \\ 0 & 0 & 0 & 1 \end{bmatrix}, (I - zT : n, 1) = \begin{bmatrix} -z & -z & 0 \\ 1 & -z & -z \\ 0 & 1-2z & -z \end{bmatrix}$$

Applying equation (7.6) results in the same GF that counts $\text{Ł}_i(A)$ given in (7.5).

$$g_{A'}(z) = -\frac{\det(I - zT : n, 1)}{z \det(I - zT)} = \frac{2z - z^2}{1 - 2z - z^2 + 2z^3}. \tag{7.7}$$

Suppose we now want to know the number of solutions of length six. We compute the sixth Taylor series coefficient to find that $|\mathcal{L}(A)_6| = [z^6]g(z) = 63$.

Deriving a Recurrence Relation: We would like a way to compute $[z^i]g(z)$ that is more direct than symbolic differentiation. We describe how a linear recurrence for $[z^i]g(z)$ can be extracted from the GF. Before we describe how to accomplish this in general, we demonstrate the procedure for our example. Combining equations (7.3) and (7.7) and multiplying by the denominator, we have

$$2z - z^2 = (1 - 2z - z^2 + 2z^3) \sum_{i \geq 0} a_i z^i.$$

Expanding the sum for $0 \leq i < 3$ and collecting terms,

$$2z - z^2 = a_0 + (a_1 - 2a_0)z + (a_2 - 2a_1 - a_0)z^2 + \sum_{i \geq 3} (a_i - 2a_{i-1} - a_{i-2} + 2a_{i-3})z^i.$$

Comparing each coefficient of z^i on the left side to the coefficient of z^i on the right side, we have the set of equations

$$a_0 = 0$$
$$a_1 - 2a_0 = 2$$
$$a_2 - 2a_1 - a_0 = -1$$
$$a_i - 2a_{i-1} - a_{i-2} + 2a_{i-3} = 0, \quad \text{for } i \geq 3$$

One can see that this results in the same solution given in equation (7.1).

This idea is easily generalized. Recall that $g(z) = p(z)/q(z)$ for finite degree polynomials p and q. Suppose that the maximum degree of p and q is m. Then

$$g(z) = \frac{b_m z^m + \ldots + b_1 z + b_0}{c_m z^m + \ldots + c_1 z + c_0} = \sum_{i \geq 0} a_i z^i.$$

Multiplying by the denominator, expanding the sum up to m terms, and comparing coefficients we have the resulting system of equations which can be solved for $\{a_i : 0 \leq i \leq m\}$ using standard linear algebra:

$$\sum_{j=0}^{i} c_j a_{i-j} = \begin{cases} b_i, & \text{for } 0 \leq i \leq m \\ 0, & \text{for } i > m \end{cases}$$

For any DFA A, since each coefficient a_i is associated with $|\mathcal{L}(A)_k|$, the recurrence gives us an $O(kn)$ method to compute $|\mathcal{L}(A)_k|$ for any string length bound k. In addition, standard techniques for solving linear homogeneous recurrence relations can be used to derive a closed form solution for $|\mathcal{L}(A)_i|$ [51].

Counting All Solutions within a Given Bound: The above described method gives a generating function that encodes each $|\mathcal{L}(A)_i|$ *separately*. Instead, we seek a generating function that encodes the number of *all solutions within a bound*. To this end we define

the automata model counting function

$$\#f_A(k) = \sum_{i \geq 0}^{k} |\mathcal{L}(A)_i|. \tag{7.8}$$

The method described above computes $\#f_A(k)$, the number of string solutions of length *exactly* k. It is of interest to compute $\#F_A(k)$, the number of solutions *within* a given bound. This is accomplished easily by using a common "trick" that is often used to simplify graph algorithms. We add a single λ-cycle (the dashed transition in Figure 7.1b) to the accepting state of the augmenting DFA A'. Then $\mathcal{L}(A')_{k+1} = \bigcup_{i=0}^{k} \mathcal{L}(A)_i \cdot \lambda^{k-i}$ and the accepting paths of strings in $\mathcal{L}(A')_{k+1}$ are in one-to-one correspondence with the accepting paths of strings in $\bigcup_{i=0}^{k} \mathcal{L}(A)_i$. Consequently, $|\mathcal{L}(A')_{k+1}| = \sum_{i=0}^{k} |\mathcal{L}(A)_i|$. Then one can see that $\#F_A(k) = \#f_{A'}(k+1)$, and so we apply the transfer matrix method on A'.

Let T be the transfer matrix of a DFA A. Another way to compute the number of paths of length k accepted by A can be computed using matrix multiplication. We compute uT^kv, where u is the row vector such that $u_i = 1$ if and only if i is the start state and 0 otherwise, and v is the column vector where $v_i = 1$ if and only if i is an accepting state and 0 otherwise. Matrix multiplication based counting method is also parameterized in the following sense: after a constraint is solved, we can count the number of solutions of any desired size k by computing uT^kv, without re-solving the constraint. The matrix multiplication method described above computes $\#f_A(k)$. We can use the same λ-cycle trick to compute $\#F_A(k)$.

The matrix multiplication method relies on computing uT^kv and so we seek to implement an efficient method for computing this product. The time and space complexity trade-offs between various methods of computing uT^kv for counting are well-studied [49, 50]. We note that one may compute T^k using matrix-matrix multiplication

with successive squaring, or one may perform left-to-right vector-matrix multiplication. While successive squaring has a better worst-case time complexity bound, we found that due to typically high sparsity of DFA transfer matrices, it is both faster and less memory intensive to use repeated vector-matrix multiplication.

First, we observe that in practice, the transfer matrix of the solution DFA is typically sparse–nodes are typically connected to very few other nodes. One may compute T^k first using matrix exponentiation and then multiply by u and v. T^k is computed by a method of successive squaring, using the recurrence $T^k = (T^{k/2})^2$ if k is even and $T^k = (T^{(k-1)/2})^2 + T$ if k is odd. This gives us an $O(n^\alpha \log_2 k)$ algorithm, where $O(n^\alpha)$ is the complexity of performing a single matrix multiplication. Typically, $\alpha = \log_2 7$, using a practical implementation of Strassen's algorithm. The upshot of this method is that it requires only logarithmically many matrix-matrix multiplications. However, even for reasonably small bounds, the resulting matrix T^k will contain very large values. In addition, T^k, as well as many intermediate matrix powers, will become dense. That is, many steps require operations on quadratically many large integers.

Rather than performing successive squaring, the value of uT^kv may simply be computed left to right: $uT^kv = (uT)T^{k-1}v$. This prevents us from using a divide and conquer technique, but with the benefit that at each step we are multiply a $1 \times n$ vector by a sparse $n \times n$ matrix. Hence, we need only keep track of the sparse matrix T and a single n-dimensional vector of large integers at each step. In our exploration of model counting algorithms for DFA, we have found this to be the best approach.

We have shown model counting methods for counting strings of a given length. The same methods allows us to perform model counting for linear constraints as well. However, we must interpret the bound k in a slightly different manner. A solution DFA $A_{\mathbb{Z}}$ for a set of integer tuples encodes the solutions as bit-strings. Thus, paths of length k in an integer automaton correspond to bit string of length k. Since we are using a 2's

complement representation with leading sign bits, bit strings of exactly length k correspond to integers in the range $[-2^{k-1}, 2^{k-1})$. Thus, the transfer matrix method allows us to perform model counting over integer domains parameterized by intervals of this form by computing $\#f_{A_{\mathbb{Z}}}(k)$. To count models for arbitrary intervals (a, b), we intersect $A_{\mathbb{Z}}$ with the DFA representing $a \leq x_i \leq b$ for any variable x_i, and then count paths in the resulting DFA.

The methods described above allow us to compute $\#F_{A_{\mathbb{S}}}(k)$ and $\#f_{A_{\mathbb{Z}}}(k)$ independently. Now, we can compute $\#\varphi(b_S, b_{\mathbb{Z}}) = \#F_{A_{\mathbb{S}}}(b_S) \cdot \#f_{A_{\mathbb{Z}}}(b_{\mathbb{Z}})$ for model counting mixed constraints.

Chapter 8

ABC Tool

In this chapter we discuss the tool ABC (Automata-Based model Counter) that implements the automata construction and model counting techniques described in this dissertation. We refer to the implementation of the algorithms described in Chapter 4 as ABC single-track and we refer the implementation of the algorithms described in Chapter 5 as ABC multi-track. We demonstrate the effectiveness of our approaches on a large set of string and numeric constraints extracted from real-world web applications. ABC source code is available online [1] along with the experimental data.

8.1 Architecture

Figure 8.1 represents high-level architecture of ABC. We can divide ABC into two main components: 1) A compilation module which performs syntactic operations, 2) automata constructor module for constraint solving and model counting.

ABC aims to supports SMTLIB language specification as an input language in order to support different types of symbolic execution tools. However, there is no standard

[1] https://github.com/vlab-cs-ucsb/ABC

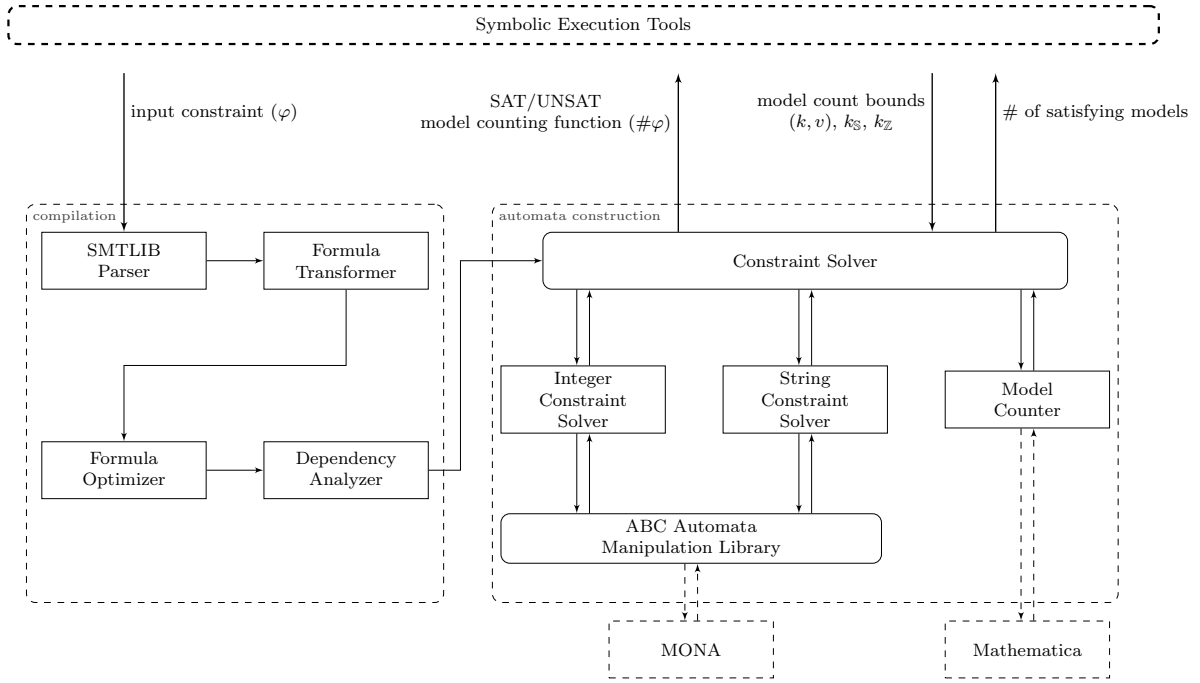

Figure 8.1: ABC architecture.

language syntax for string theory in SMTLIB specifications. Hence, ABC follows the syntax that CVC4 SMT solver uses for the string theory [52]. Once an input constraint is given to ABC it first parses it and *Formula Transformer* pushes negations down into the boolean connectives. It also checks for syntactic level optimizations that can be done such as constant folding and constant propagation. Next, *Formula Optimizer* optimizes the input formula based on equivalence relations. It also generates implied numeric constraints for the string constraints that have non-regular truth sets. Next, *Dependency Analyzer* checks for the dependencies between variables and divides the input constraint into groups that does not share any common variable. At the end of compilation phase an Abstract Syntax Tree (AST) of the input constraint along with the additional information on variables are passed to automata-construction module. *Constraint Solver* is responsible for managing automata construction for different theories. *Integer Constraint Solver* and *String Constraint Solver* modules implements the algorithms described

in Chapter 5. ABC also provides an automata manipulation library that models string operations from different programming languages. ABC automata manipulation library is extended from LibStranger[2] library and depends on Mona[3] automata manipulation library. *Model Counter* implements the automata-based model counting algorithms described in Chapter 7. We specifically added support for matrix exponentiation based model counting using big number libraries available to C++ and symbolic model counting using Mathematica[4].

We used C++ as a main development language. ABC is implemented as an *autotools* project to support portability among different systems. It can be installed as an executable or as a C++ shared library. ABC also provides a Jni interface which makes it easily available for Java applications.

8.2 Experimental Evaluation

We first evaluate ABC single-track version on a wide range of benchmarks including set of Java application benchmarks, SMTLIB translation of Kaluza JavaScript benchmarks, and several examples from the SMC (String Model Counter) distribution. Next, we evaluate ABC multi-track version on a wide range of constraints that are extracted from real world applications and compare its performance against string model counter SMC, integer model counter LattE, and ABC single-track version.

8.2.1 ABC Single-track Evaluation

In our experiments we compared model counting performance of ABC to SMC [42] and satisfiability check performance to CVC4 [52]. We ran all the experiments on an

[2] https://github.com/vlab-cs-ucsb/LibStranger
[3] https://github.com/cs-au-dk/MONA
[4] https://www.wolfram.com/mathematica/

Benchmarks	# Constraints	Frequency of Operations Per 1000 Formulas									
		match	·	=	length	replaceall	indexof	contains	begins	ends	substring
ASE	116164	0.42	386.10	129.39	382.54	639.28	4.11	7.52	16.91	7.51	41.17
Kaluza Small	368433	30.29	93.89	224.87	46.84	0	0	0	0	0	0
Kaluza Big	5138323	38.12	129.53	164.64	60.46	0	0	0	0	0	0

Table 8.1: Constraint characteristics.

Intel I5 machine with 2.5GHz X 4 processors and 32 GB of memory running Ubuntu 14.04[5].

Table 8.1 shows the frequency of string operations from our string constraint grammar that are contained in the ASE, Kaluza Small, and Kaluza Big benchmark sets. ASE benchmarks are from Java programs and represent server-side code [53]. The Kaluza benchmarks are taken from JavaScript programs and represent client-side code [54]. All three benchmarks have regular expression membership ($\in$), concatenation (.), string equality (=), and length constraints. However, the ASE benchmark contains additional string operations that are typically used for input sanitization, like replaceall and substring.

Java Benchmarks. String constraints in these benchmarks are extracted from 7 real-world Java applications: Jericho HTML Parser, jxml2xql (an xml-to-sql converter), MathParser, MathQuizGame, Natural CLI (a natural language command line tool), Beasties (a command line game), HtmlCleaner, and iText (a PDF library) [53]. These benchmarks represent server-side code and employ many input-sanitizing string operators such as replaceall and substring as seen in Table 8.1. These string constraints were generated by extracting program path constraints through dynamic symbolic execution [53].

Java benchmarks are generated to do an empirical evaluation of several string constraint solvers. As a part of this empirical evaluation, the authors use the symbolic string analysis library of Stranger [3, 55, 29] to construct automata for path constraints

[5]Results of our experiments are available at `http://vlab.cs.ucsb.edu/ABC/`

on strings. In order to evaluate the model counting component of ABC, we ran their tool on the 7 benchmark sets and output the resulting automata whenever the constraint is satisfiable. Out of 116,164 string path constraints, 66,236 were found to be satisfiable and we performed model counting on those cases. The constraints in Java benchmarks are all single-variable or pseudo-relational constraints. The resulting automata do not have any over-approximation caused by relational constraints. As a measure of the size of the resulting automata, we give the number of BDD nodes used in the symbolic transition relation representation of MONA. The average number of BDD nodes for the satisfiable path constraints is 69.51 and the size of the each BDD node is 16 bytes. For these benchmarks our model-counter is efficient; the average running time of model counting per path constraint is 0.0015 seconds and the resulting model-counting recurrence is precise, i.e., gives the exact count for any given bound.

SMC is not able to handle the constraints in this data set since it does not support sanitization operations such as replaceall.

SMC Examples. For a comparative evaluation of our tool with SMC, we used the examples that are listed on SMC's web page. We translated the 6 example constraints listed in Table 8.2 into SMTLIB language format that we support. We inspected the examples to confirm that they have regular truth sets, i.e., our analysis generates a precise model-counting function for these constraints. We compare our results with the results reported in SMC's web page. The first column of the Table 8.2 shows the file names of these example constraints. The second column shows the bounds used for obtaining the model counts. The next two columns show the log-scale SMC lower and upper bound values for the model counts. The last column shows the log-scale model upper bound value produced by ABC. We omit the decimal places of the numbers to fit them on the page. For all the cases ABC generates a precise count given the bound. ABC's count is exactly equal to SMC's upper bound for four of the examples and is

Benchmark	**bound**	$l_{\mathbf{smc}}$	$u_{\mathbf{smc}}$	$u_{\mathbf{abc}}$
nullhttpd	500	3752	3760	3760
ghttpd	620	4880	4896	4896
csplit	629	4852	4921	4921
grep	629	4676	4763	4763
wc	629	4281	4284	4281
obscure	6	0	3	2

Table 8.2: Log scaled ABC upper bound (u_{abc}) and SMC lower and upper bounds (l_{smc}, u_{smc}) comparison on SMC case studies.

exactly equal to SMC's lower bound for one example. For the last example ABC reports a count that is between the lower and upper bound produced by SMC. Note that these are log scaled values and actual differences between a lower and an upper-bound values are huge. Although SMC is unable to produce an exact answer for any of these examples, ABC produces an exact count for each of them.

JavaScript Benchmarks. We also experimented with Kaluza benchmarks which were extracted from JavaScript code via dynamic symbolic execution [54]. These benchmarks are divided to a small and large set based on the sizes of the constraints. These benchmarks have been used by both SMC and CVC4 tools. ABC handles 19,731 benchmark constraints in the satisfiable small set with an average of 0.32 seconds per constraint for model counting, whereas SMC handles 17,559 constraints with an average of 0.26 seconds per constraint. ABC handles 1,587 benchmark constraints in satisfiable big set with an average of 0.34 seconds per constraint for model counting, whereas SMC handles 1,342 constraints with an average of 5.29 seconds per constraint. We were not able to do a one-to-one timing and precision comparison between ABC and SMC for each constraint due to an error in the SMC data file (the mapping between file names and results is incorrect).

Satisfiability Checking Evaluation. We ran ABC on SMTLIB translation of the full set of JavaScript benchmarks. We put a 20-second CPU timeout limit on ABC for each benchmark constraint. Table 8.3 shows the comparison between ABC and the

Benchmarks	sat-sat	unsat-unsat	sat-unsat	unsat-sat	sat-timeout
sat/small	19728	3	0	0	0
sat/big	1587	0	0	0	0
unsat/small	8139	3013	74	0	0
unsat/big	3419	5904	2385	0	2359

Table 8.3: Constraint-solver comparison between ABC and CVC4.

CVC4 [52] constraint solver based on the CVC4 results that are available online. The first column shows the initial satisfiability classification of the data set by the creators of the benchmarks [54]. The next two columns show the number of results that ABC and CVC4 agree. At each column header, left hand side corresponds to ABC's answer and right hand side corresponds to CVC4's answer (e.g., unsat-sat means ABC returns unsat and CVC4 returns sat for the same constraint). The last three columns show the cases where ABC and CVC4 differ. Note that, since ABC over-approximates the solution set, if the given constraint is not single-valued or pseudo-relational, it is possible for ABC to classify a constraint as satisfiable even if it is unsatisfiable. However, it is not possible for ABC to classify a constraint unsatisfiable if it is satisfiable. Out of 47,284 benchmark constraints ABC and CVC4 agree on 41,793 of them. As expected ABC never classifies a constraint as unsatisfiable if CVC4 classifies it as satisfiable. However, due to over-approximation of relational constraints, ABC classifies 2,459 constraints as satisfiable although CVC4 classifies them as unsatisfiable. A practical approach would be to use ABC together with a satisfiability solver like CVC4, and, given a constraint, first use the satisfiability solver to determine the satisfiability of the formula, and then use ABC to generate its truth set and the model counting function.

The average automata construction time for big benchmark constraints is 0.44 seconds and for small benchmark constraints it is 0.01 seconds. CVC4 average running times are 0.18 seconds and 0.015 seconds respectively (excluding timeouts). CVC4 times out for 2359 constraints, whereas ABC never times out. For those 2359 constraints, ABC

reports satisfiable. ABC is unable to handle 672 constraints; the automata package we use (MONA) is unable to handle the resulting automata and we believe that these cases can be solved by modifying MONA. For these 672 constraints; CVC4 times out for 29 of them, reports unsat for 246 of them, and reports sat for 397 of them. There are also a few thousand constraints from the Kaluza benchmarks that CVC4 is unable to handle.

8.2.2 ABC Multi-track Evaluation

We compare ABC with two existing model counters: (1) SMC [42], a string model counter, and (2) LattE [56, 57], a linear integer arithmetic solver with model counting capabilities. All experiments were run on an Intel i5 machine with 2.5GHz X4 processors and 32GB of memory running Ubuntu 14.04.

ABC-SMC Comparison for String Constraints: We ran ABC on two benchmarks of satisfiable constraints which were generated via symbolic execution of JavaScript and originally solved with the Kaluza string solver[54]. The authors of SMC translated these benchmarks into their input format and separated them into two sets: SMCSmall and SMCBig. We translated them from SMC format to ABC input format. The SMCSmall set contains 17533 test constraints and SMCBig contains 1327 test constraints. ABC gives an upper bound on the model count for all tuples of string variables, while SMC gives both a lower and upper bound. We compare two versions of ABC against the upper bound SMC reports: the first version, ST ABC with only single-track automata, implements the techniques described in [58] and the second version, MT is ABC with multi-track automata that supports both string and numeric formulae based on the constructions we discussed in this thesis. Table 8.4 compares the bounds given by ST to SMC while Table 8.4 compares the bounds given by MT to SMC. ST takes only 0.0023s and 0.396s per constraint for SMCSmall and SMCBig, respectively, but consistently reports

Benchmark	**#Constraints**	$u_{ST} < u_{SMC}$	$u_{ST} = u_{SMC}$	$u_{ST} > u_{SMC}$
SMCSmall	17533	1 (0.0%)	16555 (94.4%)	977 (5.6%)
SMCBig	1327	0 (0.0%)	281 (21.2%)	1046 (78.2%)
		$u_{MT} < u_{SMC}$	$u_{MT} = u_{SMC}$	$u_{MT} > u_{SMC}$
SMCSmall	17533	862 (4.9%)	16669 (95.1%)	0 (0.0%)
SMCBig	1327	1046 (78.8%)	281 (21.2%)	0 (0.0%)
		$u_{MT} < u_{ST}$	$u_{MT} = u_{ST}$	$u_{MT} > u_{ST}$
SMCSmall	17533	977 (5.6%)	16556 (94.4%)	0 (0.0%)
SMCBig	1327	1046 (78.8%)	281 (21.2%)	0 (0.0%)

Table 8.4: ABC single-track (u_{ST}), ABC multi-track (u_{MT}) and SMC (u_{SMC}) upper bounds comparison.

an upper bound showing little to no improvement compared to SMC. MT takes longer to solve for SMCBig (0.012s and 17.8s per constraint for SMCSmall, SMCBig), but gives a more precise upper bound than SMC for 4.9% of constraints in SMCSmall, and 78.8% of constraints in SMCBig. SMC takes 0.42s for SMCSmall and 4.13s for SMCBig on average.

ABC-LattE Comparison for Numeric Constraints: We compare ABC with LattE in the context of program analysis using the benchmarks (Table 8.5) from reliability analysis [59] and side-channel analysis [60, 61]. First nine applications, including sorting algorithms, are benchmarks from reliability analysis [59]. We extended the reliability analysis benchmarks by adding Merge sort, Quick sort, and Binary search examples. Password, LawDB, and CRIME are benchmarks from timing/space side-channel analysis [60, 61].

Both analysis techniques require a symbolic execution tool to extract program path constraints, and a model counting tool to enable quantitative analysis on the path constraints. The implementation of both analysis techniques uses SPF for symbolic execution. Some of the benchmarks, e.g., sorting algorithms, require a data structure with certain size in order to enable symbolic execution. We fixed the size of the such structures to 6. We collected path constraints from fifteen applications using SPF. We counted so-

Application	**#PCs**	$b_1 = 4$	$b_2 = 8$	$b_3 = 16$	$b_4 = 32$	$\{b_1, b_2, b_3, b_4\}$
Alarm	2000	+0.002	+0.003	+0.003	+0.003	+0.039
Booking	2000	+0.003	+0.003	+0.003	+0.003	+0.043
DaisyChain	1434	+0.235	+0.023	+0.023	+0.022	+0.343
FlapController	641	+0.021	+0.021	+0.021	+0.021	+0.114
RobotGame	660	+0.137	+0.130	+0.130	+0.128	+0.560
Bubble sort	720	+0.004	+0.003	+0.001	−0.004	+0.061
Heap sort	1943	+0.005	+0.005	+0.004	+0.001	+0.066
Insertion sort	720	+0.004	+0.003	+0.001	−0.005	+0.061
Selection sort	1359	+0.005	+0.005	+0.003	−0.000	+0.065
Merge sort	720	+0.004	+0.003	+0.001	−0.005	+0.060
Quick sort	1134	+0.005	+0.005	+0.003	−0.001	+0.065
Binary search	13	+0.004	+0.000	−0.003	−0.015	+0.063
Password	7	+0.044	+0.044	+0.044	+0.044	+0.208
LawDB	8	−0.003	−0.005	−0.008	−0.008	+0.018
CRIME	1540	+0.182	+0.249	+0.252	+0.245	+0.972

Table 8.5: Average time differences in seconds between ABC and LattE. b is bit-length bound for model counting. Positive means ABC is faster.

lutions to the path constraints given bit-length bounds 4, 8, 16, and 32. ABC and LattE return identical counts for all constraints in all cases as both model counters are precise in counting numeric constraints. We focus on the timing comparison between ABC and LattE.

The LattE input format only supports conjunctions of linear equalities and inequalities. In order to handle disequalities ($\neq$) that can arise from path constraints, a preprocessing step is required. LattE integration with SPF uses Omega [62] to convert disequalities into inequalities, which comes with the benefit of constraint simplifications whenever possible. LattE timing measurements includes Omega simplification time and SPF simplification time. Details of the LattE integration can be found in [59, 60, 61].

Table 8.5 shows that in general ABC performs better than LattE. As bit-length bound increases the timing differences between ABC and LattE decreases in most of the cases and LattE performs better for some of the applications with the larger bounds. As bit-length bound increases, ABC needs to perform more matrix multiplications which takes more time.

Notice that the timing difference between ABC and LattE, when we count for multiple bounds, is largest. Note that even ABC does worse than LattE in some cases, it performs better when counting is done for multiple bounds. Since ABC is a parameterized model counter, it first solves a constraints without putting any bounds on it and then reuses the generated automaton to count given multiple bounds. In contrast, LattE needs to be called separately for each bound.

LattE performs better in counting for LawDB benchmark for all individual bound. It also performs better for Binary search benchmark in general for the individual bounds. In both cases, analysis of such benchmarks requires an initial constraint on the symbolic input variables which increases the size of the formulae. The multi-track DFA generated by ABC can be exponential in the size of the input constraints which affects both constraint solving and model counting times. In contrast, LattE implements a polynomial time counting algorithm which is less sensitive to the size of the formula.

ABC Performance on Mixed String and Numeric Constraints: We evaluated ABC performance on mixed constraints that neither SMC nor LattE can handle. We created a benchmark for mixed constraints using SMCSmall benchmark. Out of 17554 test cases in SMCSmall, 6617 contained length constraints on string variables. Length constraint in SMC benchmarks contains only integer constants. For every such length constraint, we replaced the constant length with a symbolic integer, thus producing mixed constraints. We ran ABC on all 6617 such constraints, and computed a projected count similarly to the method used for the ABC-SMC comparison. ABC completed after 210 seconds (0.03 seconds per constraint) in comparison, ABC averaged 0.01 seconds per constraint for the original SMCSmall benchmark.

Increasing ABC Performance and Precision with Heuristics: The sizes of the multi-track DFA generated by ABC can be exponential in the size of the input constraints. In our experiments we always use the equivalence class generation and

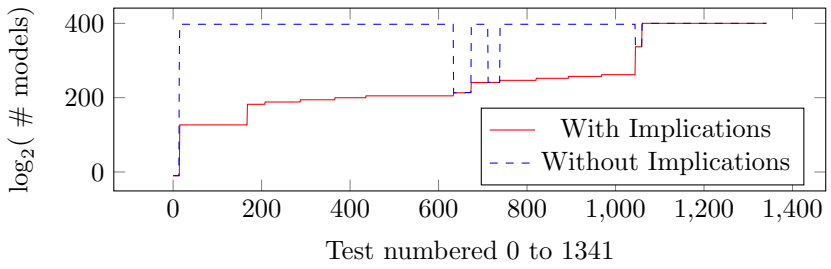

Figure 8.2: Precision differences between different configurations for ABC.

dependency heuristics, since without these heuristics ABC runs out of memory for large formulae. In order to evaluate the effectiveness of the implication heuristic, we run different versions of our tool on the SMCBig benchmark: a version with the implications heuristics and a version without. Both implementations used equivalence class generation and dependency analysis. The results given by each version are shown in figure 8.2. The version with added implications completed the benchmark after 6.60 hours (17.60 seconds per constraint), while the version without implications took 0.39 hours (1.05 seconds per constraint). Intuitively, adding implications tends to increase precision, often at the expense of longer execution times. The results reinforce this intuition, at least for this particular benchmark.

Chapter 9

Automated Test Case Generation via String Analysis

In this chapter, we present an automated testing framework that targets testing of input validation and sanitization operations in web applications for discovering vulnerabilities. In Chapter 1, we discuss the importance of of input validation and sanitization operations for web applications. We also provided a running automata-based symbolic string analysis technique to discover security vulnerabilities that are due to incorrect input validation or sanitization.

Although static string analysis techniques are powerful, they are not always feasible for analyzing real world applications due to various reasons such as cost of the analysis, missing models for library functions, and the difficulty of statically resolving dynamic behaviors of programs written in scripting languages. Moreover, since static string analysis is undecidable, these techniques use abstractions and approximations which lead to false positives.

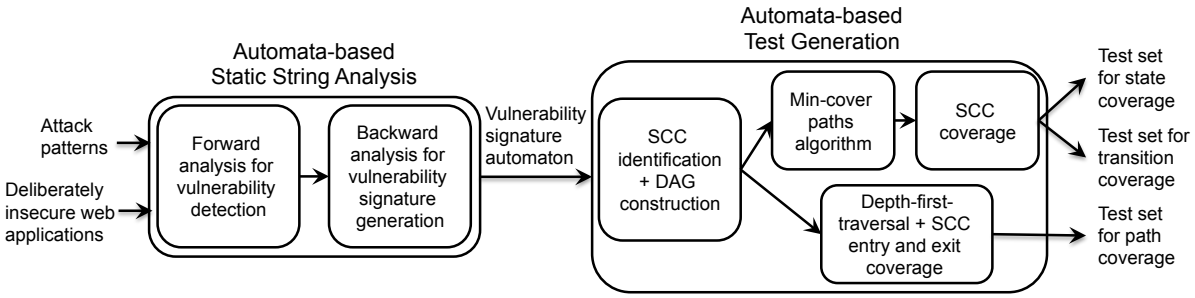

Figure 9.1: Automated Test Generation from Vulnerability Signatures.

9.1 Motivation and Overview

The high-level flow of our automated testing framework for input validation and sanitization functions is shown in Figure 9.1. This section explains different aspects of our approach, before explaining the technical details in the following sections.

Our framework combines automated testing techniques with static string analysis techniques for vulnerability analysis [63]. We use static string analysis to obtain an over-approximation of all the input strings that can be used to exploit a certain type of vulnerability. This set of strings is called a vulnerability signature, which could be an infinite set containing arbitrarily long strings.

For specification of different types of vulnerabilities we use attack patterns developed by security researchers. These are regular expressions that characterize the strings that would cause a vulnerability when sent to a security sensitive function. Given an attack pattern and a web application, we use automata-based string analysis techniques to generate an automaton that corresponds to the vulnerability signature for that application for the type of vulnerability characterized by the attack pattern. As input web applications, we use the deliberately insecure web applications that are developed by security researchers to demonstrate different types of programming practices that lead

to vulnerabilities.

Using the vulnerability signature automata generated by analyzing the deliberately insecure web applications, we automatically generate test cases based on three coverage criteria: state, transition and path coverage. Each test case corresponds to a string such that, when that string is given as a text field input to a web application, it may exploit the vulnerability that is characterized by the given vulnerability signature. Our automated test generation algorithm tries to minimize the number of test cases while achieving the given coverage criteria.

9.1.1 Automata-based Static String Analysis

Our automated testing framework generates test cases from vulnerability signatures. A *vulnerability signature* is a characterization of all user inputs that can exploit a vulnerability. In our framework we use automata-based string analysis in which vulnerability signatures are represented as automata. Automata-based string analysis is a static program analysis technique. Given a set of input values represented as automata, it symbolically executes the program to compute the set of string values that can reach to each program point. Using a forward-analysis that propagates input values to sinks (i.e., security sensitive functions), it is possible to identify attack strings that can reach to a given sink. Then, a backward analysis that propagates the attack strings back to user input results in an automaton that corresponds to the vulnerability signature.

9.1.2 Generating Vulnerability Signatures from Deliberately Insecure Applications

Security researchers have developed applications that are deliberately insecure to demonstrate typical vulnerabilities. These applications are sometimes used to teach dif-

ferent pitfalls to avoid in developing secure applications, and sometimes they are used as benchmarks for evaluating different vulnerability analysis techniques. In our framework we use static string analysis techniques to analyze deliberately insecure applications and to compute a characterization of inputs that can exploit a given type of vulnerability.

In order to generate the vulnerability signature for an application, we need an attack pattern (specified as a regular expression) that characterizes a particular vulnerability. An attack pattern represents the set of attack strings that can exploit a particular vulnerability if they reach a sink (i.e., a security sensitive function). Attack patterns for different types of vulnerabilities are publicly available and can be used for vulnerability analysis.

Given an attack pattern and a deliberately insecure web application, we use automata-based static string analysis techniques to generate a vulnerability signature automaton that characterizes all the inputs for that application that can result in an exploit for the vulnerability characterized by the given attack pattern. I.e., the vulnerability signature automaton only accepts the strings that are in the vulnerability signature. In the next phase of our approach we automatically generate test cases from the vulnerability signature automaton.

9.1.3 Automated Test Generation from Vulnerability Signatures

Given a vulnerability signature automaton, any string accepted by the automaton can be used as a test case. Hence, any path from the start state of the vulnerability signature automaton to an accepting state characterizes a string which can be used as a test case. However, a vulnerability signature automaton typically accepts an infinite number of strings since, typically, there are an infinite ways one can exploit a vulnerability. In order to use vulnerability signature automata for testing, we need to somehow prune

this infinite search space. Our overall goal is to minimize the number of test cases while making sure that we cover all possible ways of exploiting a vulnerability.

The mechanism that allows an automaton to represent an infinite number of strings is the loops in the automaton. So, in order to minimize the number of test cases, we have to minimize the way the loops are traversed. We do this by identifying all the strongly-connected components (SCCs) in an automaton and then collapsing them to construct a directed acyclic graph (DAG) that only contains the transitions of the automaton that are not part of an SCC and represents each SCC as a single node. Using this DAG structure, we do test generation for three coverage criteria: 1) *state coverage* where the goal is to cover all states of the automaton (including the ones in an SCC), 2) *transition coverage*, where the goal is to cover all transitions of the automaton (including the ones in an SCC), 3) *path coverage*, where the goal is to cover all the paths in the DAG that is constructed from the automaton, while also covering all possible ways to enter and exit from an SCC.

We implement the state and transition coverage using the min-cover paths algorithm that we execute on the DAG representation followed by a phase where we ensure the coverage of the states and transitions inside the SCC nodes. We implement the path coverage using depth-first-traversal, where, when an SCC node is encountered, we ensure that all entry and exit combinations are covered in the generated test cases.

9.1.4 A Sanitization Example

One of the well-known XSS attack strings is the following:

```
<script>alert('XSS')</script>
```

The script-tag indicates executable code and a malicious user might be trying to store a malicious script to be executed on another user's machine later on. Now, consider the

```
1  <?php
2   if(!array_key_exists ("name", $_GET) || $_GET["name"] == NULL || $_GET["
       name"] == "") {
3     $isempty = true;
4   } else {
5     $html .= "<pre>";
6     $html .= "Hello ";
7     $html .=  str_replace( "<script>", "", GET["name"]);
8     $html .= "</pre>";
9   }
10 ?>
```

Figure 9.2: A sanitization example.

example code in Figure 9.2 extracted from a deliberately insecure web application. This code is sanitizing the input provided by the user for the "name" field in line 7 by deleting all appearances of the string `<script>` (it deletes it by replacing each appearance of the string `<script>` with the empty string). Later on in the program, the variable `$html` is used as an input for a security sensitive function, so if the sanitization is not done properly this application would have a vulnerability.

We can try to check if the application is vulnerable by testing it with the above attack string. As expected the sanitization code will correctly remove the script-tag and sanitized input will be `alert('XSS')</script>`. So, this test input does not detect a vulnerability. However, this application has a vulnerability and the sanitization used in Figure 9.2 is incorrect.

One can generalize the attack strings for the XSS vulnerability as an attack pattern using the following regular expression:

`/.*<script.*>.*/`

When we run the automata-based string analysis on the example shown in Figure 9.2, we find out that the intersection of the set of strings that can reach the sink and the above attack pattern is not empty, i.e., there are some inputs that will cause a string

containing the script-tag reach the sink. So, we generate the vulnerability signature for this application which results in an automaton that contains 59 states and 8530 transitions. Note that, this vulnerability signature automaton captures the fact that the string-replace operation in line 7 will delete all appearances of the string `<script>` from the input. The reason that there are thousands of transitions is due to the fact that there is a transition for each ASCII character from each state.

When we use our automated test generation technique to generate a test string from the vulnerability signature automaton, we obtain the following test input:

`<scrip<script>t>`

When we run the application with this input we discover an attack, i.e., the sink function receives an input that contains the string `<script>`. This is due to the fact that the incorrect sanitization function in Figure 9.2 deletes the substring `<script>` from the above test input and creates the attack string.

In our framework, we use the test strings generated from vulnerability signatures of deliberately insecure web applications to test other applications. If the applications we test contain sanitization errors similar to the errors in deliberately insecure web applications or if they do not use proper sanitization, then the generated test cases can discover their vulnerabilities without analyzing them statically. Note that the test inputs generated from vulnerability signatures can also be used for applications that are statically analyzable in order to eliminate false positives and construct exploits (i.e., to generate concrete inputs that demonstrate how a vulnerability can be exploited).

9.2 Converting Vulnerability Signature Automata to DAGs

Vulnerability signatures represents all the possible user inputs to an input field that can exploit an attack.

A vulnerability signature V is a DFA such that $V = (Q, \Sigma, \delta, q_0, F)$, where Q is the set of states, Σ is the input alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. Each transition $t \in \delta$ is a tuple $t = (q, c, q')$ where $q = \text{source}(t)$, $q' = \text{target}(t)$ and $c \in \Sigma$.

A naive way to generate accepting strings (test cases) from a vulnerability signature automaton is to exploring all possible accepting paths. If the alphabet size is large, the number of transitions between two states can cause

an exponential blow up in the number of accepting paths in the automaton, and this leads to a large search space for test generation. As an example, assume that the alphabet Σ is ASCII alphabet. Consider state q_2 in Figure 9.3. For this relatively small automaton there are 128×128 accepting paths. Such cases can be common in vulnerability signatures. Our solution to this problem is to collapse the transitions that have the same source and target states into one transition as shown in Figure 9.4. The label of the collapsed transition is a range of characters corresponding to each transition that it represents. During test generation we only pick one character from the range representing the all corresponding transitions. This allows us to avoid exponential blow up in the number of accepting paths. For the rest of the chapter we assume that all transitions with the same source and target states are collapsed.

Another difficulty with vulnerability signature automata is that they can contain cycles which results in an infinite number of accepting paths. As an example, in Figure 9.5, states $\{q_1, q_2, q_3\}$ and $\{q_4, q_5\}$ form cycles. In order to bound the number of

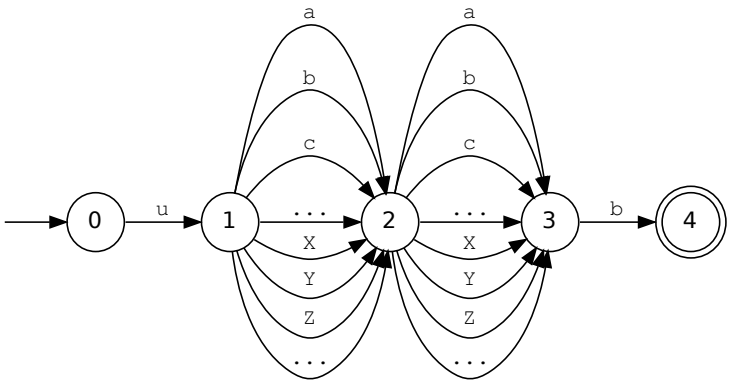

Figure 9.3: Example case for large number of paths.

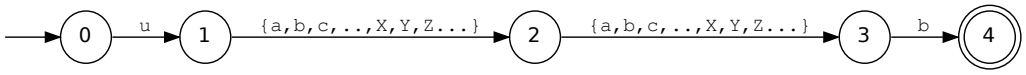

Figure 9.4: Example of collapsed transitions.

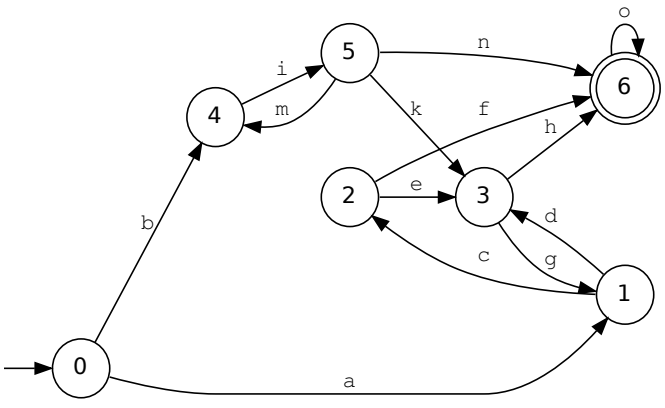

Figure 9.5: Example of cycles in an automaton.

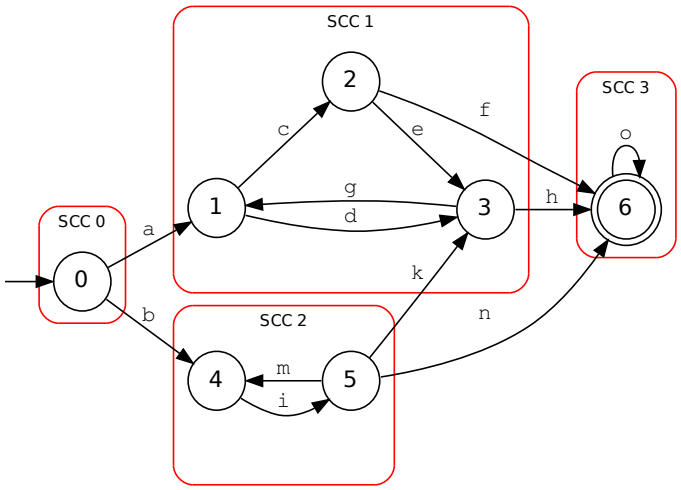

Figure 9.6: High level DAG representation.

accepting paths and, therefore the search space for test generation, we extract a high level representation of the given vulnerability signature automaton by identifying its strongly connected components (SCC). The high level representation we obtain is a directed acyclic graph DAG $= (N, E)$ where N is the set of SCCs and E is the set of edges between SCCs. At the automaton level each edge $e \in E$ is a transition such that source$(e) \in \text{scc}_x$, target$(e) \in \text{scc}_y$ and $\text{scc}_x \neq \text{scc}_y$. We use Tarjan's strongly connected components algorithm to identify the cycles in the vulnerability signature automata [64]. The worst case time complexity of this algorithm is $O(|Q| + |\delta|)$. High-level DAG representation for the automaton in Figure 9.5 is shown in Figure 9.6. It consists of four strongly connected components $N = \{\text{SCC}_0, \text{SCC}_1, \text{SCC}_2, \text{SCC}_3\}$, and six edges among them $E = \{e_a, e_b, e_k, e_n, e_f, e_h\}$.

9.3 State and Transition Coverage for Vulnerability Signature Automata Using Min-Cover Paths Algorithm

In this section we discuss generating test cases from vulnerability signature automata based on state and transition coverage criteria. Given a vulnerability signature automaton V, let $\mathcal{L}(V)$ denote the set of strings accepted by V. Our aim is to find two sets of strings $S_{\text{sc}}, S_{\text{tc}} \subseteq \mathcal{L}(V)$ that achieve state and transition coverage, respectively. The state and transition coverage definitions are as follows:

- For each state in $q \in Q$ there must be at least one string in S_{sc} such that the accepting path for that strings visits q.
- For each (collapsed) transition $t \in \delta$ there must be at least one string in S_{tc} such that the accepting path for that string includes t.

Finally, we want to generate the sets S_{sc} and S_{tc} in such a way that $|S_{\text{sc}}|$ and $|S_{\text{tc}}|$ are minimized.

The problem of finding minimum number of strings based on state and transition coverage criteria is very similar to a well-known graph problem called *minimum cover paths*. Given a directed acyclic graph, minimum cover paths is the least number of paths that visits each edge of the graph at least once. Minimum cover paths problem has been studied in different research areas and there are well known solutions to this problem [65, 66]. One known solution is to reduce minimum cover paths problem to the minimum flow problem [65, 67, 66]. We follow this basic approach with some modifications. We can divide the state and transition coverage algorithms into five main steps: 1) Initialization of DAG, 2) Converting DAG into a flow network, 3) Minimum flow algorithm, 4) Finding

minimum covering paths, 5) Extending paths with SCC Coverage.

9.3.1 Initialization of DAG

A vulnerability signature automaton V has one start state q_0 and a set of final states F. In order to apply flow algorithms and minimum covering paths algorithm, one virtual final state q_v is added to Q, for each $q \in F$, a virtual transition $t_v = (q, \lambda, q')$ is added to the transition relation δ where $\lambda \notin \Sigma$. The modified automaton has one start state q_0 and one final state q_v. A DAG representation DAG $= (N, E)$ is constructed from the modified automaton as described in the previous section. We use $n_0 \in N$ to denote the start node of the DAG where $n_0 = \text{SCC}_0$ and $q_0 \in \text{SCC}_0$. Similarly, we use $n_v \in N$ to denote the as final node of the DAG such that $n_v = \text{SCC}_v$ and $q_v \in \text{SCC}_v$.

A vulnerability signature automaton always has a sink state that terminates non-accepting paths corresponding to non-accepting strings. As a result, corresponding DAG representation has a sink node that does not have any outgoing edges. We generate only the strings that are accepted by vulnerability signature automaton. To do so we remove the sink node and all incoming edges to the sink node from the DAG using a depth first traversal with a worst case complexity of $O(|E|)$.

9.3.2 Converting DAG into a Flow Network

A flow network is a DAG where each edge has a capacity and each edge receives a flow. Capacity for each edge $e \in E$ is a non-negative real value $c(e) \geq 0$. Flow is a function $f : E \rightarrow R$ that satisfies the following properties:

- $\forall e \in E : f(e) \leq c(e)$.
- $\forall e \in E, e' \in E : f(e) = -f(e') \wedge \text{source}(e) = \text{target}(e') \wedge \text{target}(e) = \text{source}(e')$.

- $\forall n \in N : (\sum_{e \in \text{incoming}(n)} f(e) + \sum_{e' \in \text{outgoing}(n)} f(e')) = 0.$

Min-cover paths algorithm does not require an upper bound for the capacity of an edge, and we assume that each edge has infinite capacity. We define a flow as the number of required visits to an edge in order to take each path from the start node to the final node. To apply the min-flow algorithm, we need an initial flow assignment for each edge in the DAG. We use a pre-processing algorithm [65] to assign an initial flow to each edge based on the number of input and output edges for each node. This is a two phase algorithm that consists of a depth first traversal starting from start node (Phase 1) followed by a reverse depth first traversal (Phase 2) if necessary. The first phase of the initialization for state coverage is shown in Algorithm 13.

Algorithm 13 Phase 1 for pre-processing of state coverage

```
 1: function PreProcessRigthSC(node, queue)
 2:     updated ← false
 3:     for each edge ∈ outgoingEdges(node) do
 4:         nextnode ← targetnode(edge)
 5:         if flow(edge) = 0 then
 6:             if #incomingEdges(nextnode) = 1 or #outgoingEdges(nextnode) = 1 then
 7:                 flow(edge) ← 1, updated ← true
 8:             else
 9:                 removeFromDag(edge)
10:             end if
11:         end if
12:     end for
13:     if ¬updated ∨ balanced(node) = 0 then
14:         return
15:     end if
16:     if updated and balanced(node) < 0 then
17:         queue.enqueue(node)
18:     else if updated and balanced(node) > 0 then
19:         distributeFlowsEvenly(node)
20:     end if
21:     for each edge ∈ outgoingEdges(node) do
22:         nextnode ← targetnode(edge)
23:         PreProcessRigth(nextnode, queue)
24:     end for
25: end function
```

The statement at line 6 checks for the edges that can be removed safely. For example

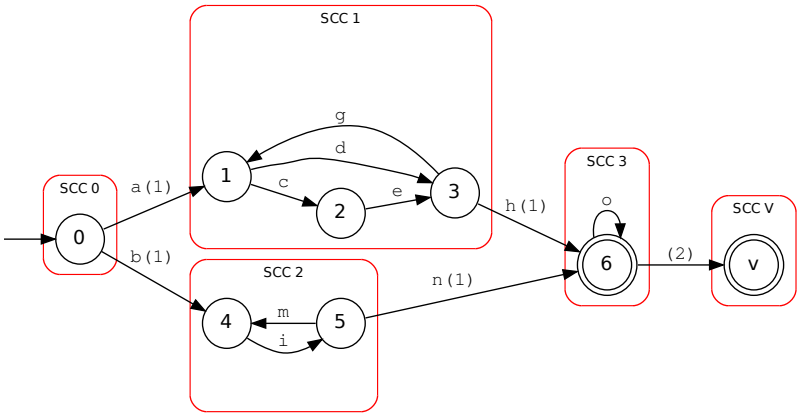

Figure 9.7: Initialized DAG for state coverage.

edges labeled with $'f'$ and $'k'$ can be safely removed from Figure 9.6. The resulting high level DAG is shown in Figure 9.7. Depending on the order that for loop retrieves the edges at line 3, algorithm may remove different edges at different runs. However, this does not affect the state coverage.

We can define the flow function flow(e) as number of visits for an edge $e \in E$. The function balanced compares the total input flow and total output flow for a node $n \in N$ based on flows for each incoming and outgoing edges. A positive balance means that the total input flow is larger than the total output flow. In that case line 19 distributes the input flows to the the output flows by updating the flow values of outgoing edges. For the case of a negative balance value, distribution is done in the reverse direction after Phase 1 finishes as described in [65]. Figure 9.7 also shows the initial flow values that are assigned to the example DAG. For the example shown in Figure 9.7, reverse pre-processing (Phase 2) is not necessary since in the first phase flows are already distributed correctly.

Phase 1 of the pre-processing algorithm for transition coverage is shown in Algorithm 14. The only modification compared to the algorithm shown in Algorithm 13 is inside the if block at line 5. The resulting flows for transition coverage are shown in Figure 9.8. Starting from the initial node, the algorithm first assigns a flow value of 1

to the edges $'a'$ and $'b'$. When it comes to SCC_2 during depth first traversal, it first assigns a flow of 1 to the edges $'k'$ and $'n'$. As a result balance value of SCC_2 becomes -1 and that SCC_2 is queued for reverse pre-processing. Similarly when algorithm first visits the SCC_1 using edges $'a'$ or $'k'$, balance value for SCC_1 becomes negative and SCC_1 is also queued for reverse pre-processing. However, when the algorithm visits SCC_1 for the second time, balance value becomes 0 and reverse pre-processing on SCC_1 does not have any effect.

Algorithm 14 Phase 1 for pre-processing of transition coverage

```
 1: procedure PreProcessRigthTC(node, queue)
 2:     updated ← false
 3:     for each edge ∈ outgoingEdges(node) do
 4:         nextnode ← targetnode(edge)
 5:         if flow(edge) = 0 then
 6:             flow(edge) ← 1, updated ← true
 7:         end if
 8:     end for
 9:     if ¬updated ∨ balanced(node) = 0 then
10:         return
11:     end if
12:     if updated ∧ balanced(node) < 0 then
13:         queue.enqueue(node)
14:     else if updated and balanced(node) > 0 then
15:         distributeFlowsEvenly(node)
16:     end if
17:     for each edge ∈ outgoingEdges(node) do
18:         nextnode ← targetnode(edge)
19:         PreProcessRigth(nextnode, queue)
20:     end for
21: end procedure
```

9.3.3 Minimum Flow Algorithm

After we have initial flows calculated, Ford-Fulkerson algorithm is applied to the flow network with some modifications [68, 65]. Modified Ford-Fulkerson algorithm computes the minimum flows to visit each transition at least once. The algorithm finds paths from the start node to the final node and removes the maximum amount of flow from each

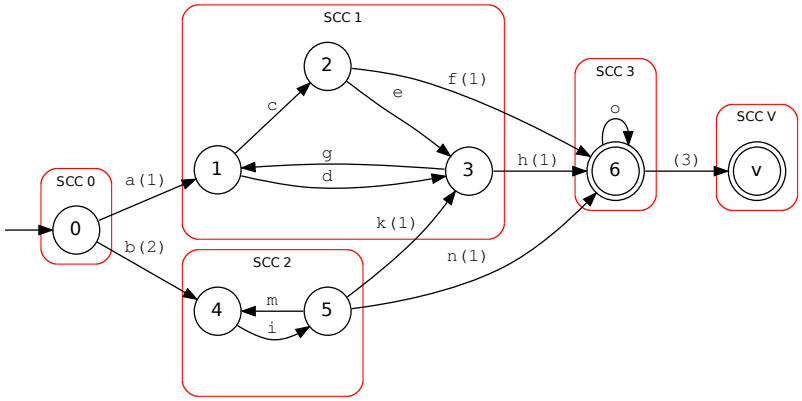

Figure 9.8: Initialized DAG for transition coverage.

path without reaching 0. Assume that our initialization phase calculated the flow for the path "bkh" in Figure 9.8 as "$b(4)k(3)h(3)$" instead of "$b(2)k(1)h(1)$". We can take away 2 flows from all the edges in the path "bkh". Time complexity of the algorithm for a DAG is $O(|p_{max}| \cdot (f_0 - f_{min}))$ where $|p_{max}|$ is the maximum length path from start node to final node, f_0 is initial flow set and f_{min} is the minimum flow [65].

9.3.4 Finding Minimum Covering Paths

After running Minimum Flow Algorithm we can start looking for minimum covering paths. Minimum Covering Paths algorithm finds the edges that have flow$(e) > 0$ and forms a path that ends at the final node (i.e., the virtual node). Algorithm 15 shows the general loop and the recursive path finding function. For example, given the DAG shown in Figure 9.8, the minimum covering paths for transition coverage are computed as: "afe_v", "$bkhe_v$", and "bne_v" where e_v is the virtual edge.

Let N_k be the set of nodes that are k edges away from the start node. Let E_k be the set of edges between N_k and N_{k+1}. Let E_{max} be the edge set with maximum size among the sets $E_0, E_1, E_2, ...E_n$. Finally, let P_{max} be the maximum length path from start node to final node. Then, worst case time complexity for state and transition coverage

Algorithm 15 Minimum covering paths algorithm

```
1: minPaths ← [ ]
2: repeat
3:     path ← FindMinPath(node_start)
4:     minPaths.add(path)
5: until path = [ ]
6: function FindMinPath(node)
7:     for each edge ∈ outgoingEdges(node) do
8:         if flow(edge) ≠ 0 then
9:             DecreaseFlowByOne(edge)
10:            nextnode ← targetnode(edge)
11:            path = FindMinPath(nextnode)
12:            if path ≠ [ ] ∨ nextnode = node_final then
13:                path.add(edge)
14:                return path
15:            end if
16:        end if
17:    end for
18:    return [ ]
19: end function
```

is $O(|P_{\max}| \times |E_{\max}|)$ and the maximum size test set size for both coverage criteria is $O(|E_{\max}|)$ which is equal to the number of minimum covering paths. For the DAGs that are extracted from the same vulnerability signature automaton let $|E_{\max}|_{\text{sc}}$ denote the size of $E_{\max}$ for the DAG generated for state coverage and $|E_{\max}|_{\text{tc}}$ denote the size of $E_{\max}$ for the DAG generated for transition coverage. Then, we have $|E_{\max}|_{\text{sc}} \leq |E_{\max}|_{\text{tc}}$. For the sets of test cases generated for state and transition coverage (S_{sc} and S_{tc}, respectively) we have $|S_{\text{sc}}| \leq |S_{\text{tc}}|$.

9.3.5 Extending Paths with SCC Coverage

Once we have the results for minimum covering paths we do a pass on each path and extend the SCC nodes $n \in N$ that represent cycles. We can define a strongly connected component as $\text{SCC} = (Q_{\text{SCC}}, \Sigma, \delta_{\text{SCC}})$ where $Q_{\text{SCC}} \subseteq Q$ and $\delta_{\text{SCC}} \subseteq \delta$. Assume there is a state $q_x \in Q_{\text{SCC}}$ and a transition $t \in \delta$. If $q_(x) = \text{target}(t)$ and $\text{source}(t) \notin Q_{\text{SCC}}$, we say state q_x is an entry point. Similarly, assume there is an edge $q_y \in Q_{\text{SCC}}$ and a transition

$t \in \delta$. If $q_(x) = \text{source}(t)$ and $\text{target}(t) \notin Q_{\text{SCC}}$, we say state q_x is an exit point.

There are two different strategies for SCC coverage based on DAG coverage algorithm in progress. Strategy for the state coverage algorithm is the following: Starting from an entry point visit all states $q \in Q_{\text{SCC}}$ at least once and end up in an exit point. Similarly, for transition coverage starting from an entry point visit all transitions $t \in \delta_{\text{SCC}}$ at least once and end up at an exit point. If $|\delta_{\text{SCC}}|$ is greater than zero, then SCC must contain a cycle like SCC_1, SCC_2, and SCC_3 in Figure 9.6. To terminate the algorithm we keep a queue for unvisited states or unvisited transitions and use depth first search whenever necessary. Algorithm 16 shows the algorithm we use for state coverage. DFS function at line 7 starts a depth first search from the state given as its first argument and searches for the state given as its second argument without being trapped in a cycle. Once it finds the state given as its second argument, it returns a path that includes all the states it visited. Algorithm for visiting all transitions $t \in \delta_{\text{SCC}}$ is the same except we keep a queue for unvisited transitions instead of unvisited states. Both algorithms have a worst case complexity of $O(|\delta_{\text{SCC}}|^2)$ which depends on the overlapping cycles within a SCC. Worst case complexity of length of the returned path is also the same as the time complexity.

Algorithm 16 SCC coverage

```
 1: procedure VisitStates(SCC, q_entry, q_exit)
 2:     path ← [ ]
 3:     notVisited ← getAllStates(SCC)
 4:     q ← q_entry
 5:     notVisited.remove(q)
 6:     while size(notVisited) ≠ 0 do
 7:         visited ← DFS(q, notVisited.dequeu(
 8:         notVisited.removeAll(visited)
 9:         path.addAll(visited)
10:         q ← visited.last()
11:     end while
12:     if q ≠ q_exit then
13:         path.addAll(DFS(q, q_exit))
14:     end if
15:     return path
16: end procedure
```

Consider the example vulnerability signature automaton shown in Figure 9.7. Based on state coverage algorithm it can produce a path $.a.h.$ where each dot corresponds to a node in the DAG. Starting from the first dot which is actually SCC_0 we extend the path. SCC_0 returns an empty path and algorithm continues with next SCC in the path $a.h.$. SCC_1 returns ce for entry point q_1 and exit point q_3 and algorithm extends the path as $aceh.$. At the end the algorithm returns the extended path $aceh$.

9.4 Path Coverage for for Vulnerability Signature Automata Using Depth First Traversal

A straight forward definition of path coverage would result in an infinite set of test cases due to loops in automata. So, given a vulnerability signature automaton V, we define $S_{pc} \subseteq L(V)$ as follows:

- For each path p in the DAG generated from V there must be a set of strings in S_{pc} such that the accepting paths for those strings must correspond to p (i.e. they must visit the same set of SCCs in the same order), and there must be an accepting path for each combination of entry and exit nodes for all the SCCs in the path p.

Path Coverage algorithm traverses DAG representation of vulnerability signature automata using a depth-first traversal (DFT). It does not have any initialization phase. It handles SCC entry-exit point coverage during path exploration. Assume current node in the DFT is n and n corresponds to a SCC. Again assume q_x is the entry point for the SCC corresponding to node n. Path coverage algorithm calculates paths for all possible combinations of q_x with all exit points using the SCC coverage algorithm we have for transition coverage. Then, it continues to explore paths in the high level DAG representation by following exit points in a DFT manner. By doing so, path coverage algorithm

calculates all possible combinations of all entry and exit points of a SCC. The path coverage algorithm generates 5 paths for the example shown in Figure 9.8.

Based on definitions we have in previous section the time complexity for path coverage is $O(|E_{kmax}|^{P_{max}})$. Test size complexity is the same as the time complexity which is basically all paths from start node to final nodes. As a result we have the following test set size comparison for the three coverage criteria for the same vulnerability signature $|S_{sc}| \leq |S_{tc}| \leq |S_{pc}|$.

9.5 Implementation and Experiments

In order to evaluate our automated testing framework, we used a deliberately insecure web application called Damn Vulnerable Web Application (DVWA) to generate vulnerability signatures. DVWA is listed in OWASP Broken Web Applications Project which lists deliberately insecure web applications. DVWA has several SQL injection, stored XSS and reflected XSS attacks with different security levels provided by the application. Security levels are no sanitization, custom sanitization, and incorrect use of built-in sanitization functions. We generated vulnerability signatures for each attack type considering different security levels. We used the Stranger static string analysis tool [69] to generate vulnerability signatures. We ran all the experiments on an Intel I5 machine with 2.5GHz X 4 processors and 32 GB of memory running Ubuntu 12.04.

Table 9.1 shows the properties of 5 vulnerability signatures generated from DVWA. We used the following well known attack patterns for vulnerability signature generation. Attack pattern `/.*<script.*>.*/` is used for vulnerability signatures `XSS 1`, `XSS 2`, and `XSS 3`. Attack pattern `/.* or 1 = 1 .*/` is used for vulnerability signature `SQLI 1` and attack pattern `/.*' or '1' = '1 .*/` is used for vulnerability signature `SQLI 2`. The sizes of the vulnerability signature automata depend on the

complexity and number of string operations that application has on user inputs. We can see that vulnerability signatures `SQLI 1` and `XSS 1` are larger than the other three vulnerability signature automata. That is because the corresponding application code has more sanitization on user input. The application code that corresponds to vulnerability signature `SQLI 2` has no sanitization at all and the generated vulnerability signature is similar to the attack pattern. For each vulnerability signature, we can see that there is a big difference between the actual number of transitions that an automaton has and the corresponding number of collapsed transitions which allows us to reduce the sizes of the generated test sets. For a given vulnerability signature, the relation between the sizes of the test sets for different coverage criteria follows the ordering we expect where $|S_{sc}| \leq |S_{tc}| \leq |S_{pc}|$. For larger vulnerability signatures, path coverage algorithm produces a large number of strings as expected. For a given vulnerability signature, average length of the strings generated for state coverage is the smallest. Since the number of states are smaller than the number of transitions this is not surprising. The SCC coverage algorithm for state coverage produces strings with smaller lengths for most of the cases.

In order to evaluate the effectiveness of our automated test generation techniques we experimented on five open-source applications 1) PHP-Fusion v7.02.05 2 (content management system), 2) RuubikCMS v1.1.1 (website content management tool), 3) UL Forum v1.1.7 (forum application), 4) Snipe Gallery v3.1.5 (image management system), 5) PHP Server Monitor v2.0.1 (server management script). We implemented a web application driver to automatically execute the applications with the automatically generated test strings. We executed each application by assigning the automatically generated test strings to the selected vulnerable input fields. We enabled xdebug tool to record the server-side function call traces for each request that our web application driver sends. After each request, the web application driver extracts the sink function calls

Vulnerability Signature	Automaton Size				Coverage Type	#Strings	Avg. String Length
	$\#q$	$\#\delta$	$\#\delta_c$	#SCC			
SQLI 1	118	16327	574	19	State	8	39
					Transition	52	451
					Path Cov	321	437
SQLI 2	16	2649	58	3	State Cov	1	15
					Transition Cov	1	210
					Path Cov	1	210
XSS 1	100	13540	481	19	State Cov	8	31
					Transition Cov	44	312
					Path Cov	229	299
XSS 2	59	8530	237	3	State Cov	1	146
					Transition Cov	8	1,717
					Path Cov	8	1,628
XSS 3	11	1718	37	4	State Cov	1	10
					Transition Cov	2	73
					Path Cov	2	73

Table 9.1: Vulnerability signature automata details where $\#q$ is the number of states, $\#\delta$ is the number of transitions, $\#\delta_c$ is the number of collapsed transitions.

with values of parameters from the trace file. For the SQL injection attacks, each call to `mysql_query` function is treated as a sink function call. For the XSS attacks, each call to `mysql_query` function that executes `INSERT` or `UPDATE` statements is treated as sink function call. If the web application driver finds a sink function call, it checks the value of the query parameter of the sink function to confirm if it contains any type of attack.

Table 9.2 shows the effectiveness of the test sets generated using different coverage criteria on different applications. The sum of the third column and the fourth column shows the total number of test strings in a test set generated from all vulnerability signatures for a given coverage criteria. For example, there are a total of 19 test strings in the test set generated from all vulnerability signatures using the state coverage criteria. Third column shows the number of test strings that detected the vulnerability in the given application (stated in the first column), and the fourth column shows the number of test strings that missed the vulnerability. We can clearly say that path coverage and transition coverage have better detection rates than state coverage. The vulnerability

Application	Coverage Type	# Detected	# Missed	Detection Rate (%)
ulforum	State	8	11	42
	Transition	79	28	74
	Path	477	84	85
ruubik	State	4	15	21
	Transition	28	79	26
	Path	157	404	28
php_fusion	State	2	17	11
	Transition	42	65	39
	Path	235	326	42
snipe	State	8	11	42
	Transition	79	28	74
	Path	477	84	85
phpservermon	State	8	11	42
	Transition	79	28	74
	Path	477	84	85

Table 9.2: Vulnerability detection performance per application.

detection rates for the applications *php_fusion* and *ruubik* are lower compared to other three applications for each coverage criteria. This is due to the fact that these applications have more string manipulation operations than the other three. For the fields selected from other three applications we observe the same detection rates. This is due to the fact that these applications all have the same type of vulnerability.

Table 9.3 shows the vulnerability detection rates of test sets generated using different coverage criteria for each vulnerability signature. It shows the distribution of the test sets in Table 9.2 to different vulnerability signatures and different coverage criteria. Path coverage criteria has better detection rates for vulnerability signatures `XSS 1` and `SQLI 1` which are the larger vulnerability signature. For relatively small vulnerability signatures, path coverage and transition coverage detection rates are the same. Vulnerability signature `SQLI 2` has the worst detection rate. As we described previously in this section, that vulnerability signature is generated from a code that has no sanitization operations, which is not good enough for detecting attacks for applications that have some string operations. One interesting result is that state coverage for all `XSS` vulnerability

Vulnerability Signature	**Coverage Type**	**# Detected**	**# Missed**	**Detection Rate (%)**
SQLI 1	State	30	10	75
	Transition	193	67	74
	Path	1231	374	77
SQLI 2	State	0	5	0
	Transition	0	5	0
	Path	0	5	0
XSS 1	State	0	40	0
	Transition	75	145	34
	Path	553	592	48
XSS 2	State	0	5	0
	Transition	30	10	75
	Path	30	10	75
XSS 3	State	0	5	0
	Transition	9	1	90
	Path	9	1	90

Table 9.3: Vulnerability detection performance per vulnerability signature.

signatures has a detection rate 0%. The application that we used to generate the vulnerability signatures concatenates HTML tags to the user inputs. Resulting vulnerability signature may include attack strings that has no closing tag >. State coverage generates only strings that do not have closing tags, but path and transition coverage criteria are able to handle that situation by visiting more transitions.

Overall, path coverage has better detection rates as expected. Transition coverage detection rates are very close to path coverage detection rates, and transition coverage generates smaller test sets. State coverage is not effective in generating attack strings for the vulnerability signatures we used.

Chapter 10

Conclusion

String constraint solving and model counting are crucial problems in vulnerability analysis. In this dissertation, we presented a model-counting string constraint solver that, given a constraint, generates: 1) An automaton that accepts all solutions to the given string constraint; 2) A model-counting function that, given a bound, returns the number of solutions within that bound. We presented a novel approach to model counting both string and numeric constraints and their combinations.

Our constraint solving approach does not assume a finite domain size during automata construction. We say that our model counting approach is parameterized since it generates a model-counting function that takes the bound as a parameter and returns the number of solutions within that bound. Our approach is able to perform model counting for arbitrary bounds for a given constraint.

We have developed a tool called ABC that implements the automata-based constraint solving and model counting techniques. ABC is publicly available and can be integrated with symbolic execution tools via its SMTLIB interface. Our experiments on thousands of constraints extracted from real-world web applications demonstrate the effectiveness and efficiency of the proposed approach. Our experimental results indicate that, automata-

based model counting approach is as efficient and as precise as domain specific model counting methods, while it is able to handle a richer set of constraints.

We also presented an automated testing framework for testing input validation and sanitization operations in web applications. In our framework, the tests are generated from vulnerability signatures that are characterized as automata. Our experiments show that vulnerability signatures generated from deliberately insecure web applications can be used to generate effective tests for identifying vulnerabilities in other applications.

In the future, we plan to improve ABC in terms of expressiveness and precision. We plan to extend our constraint language with quantifiers over string and integer variables. We also plan to add support for regular expression variables in our constraint language. In our current implementation we use multi-track automata for the string predicate operations. We can use multi-track automata for the term operations as well by adding an output track. That would enable us to handle relational constraints with better precision in the future.

Bibliography

[1] Z. Su and G. Wassermann, *The essence of command injection attacks in web applications*, in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, (New York, NY, USA), pp. 372–382, ACM, 2006.

[2] G. Wassermann and Z. Su, *Sound and precise analysis of web applications for injection vulnerabilities*, in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, (New York, NY, USA), pp. 32–41, ACM, 2007.

[3] F. Yu, T. Bultan, M. Cova, and O. H. Ibarra, *Symbolic string verification: An automata-based approach*, in *Proceedings of the 15th International Workshop on Model Checking Software*, SPIN '08, (Berlin, Heidelberg), pp. 306–324, Springer-Verlag, 2008.

[4] F. Yu, T. Bultan, and O. H. Ibarra, *Symbolic string verification: Combining string analysis and size analysis*, in *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, TACAS '09, (Berlin, Heidelberg), pp. 322–336, Springer-Verlag, 2009.

[5] F. Yu, M. Alkhalaf, and T. Bultan, *Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses*, in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, (Washington, DC, USA), pp. 605–609, IEEE Computer Society, 2009.

[6] A. Aydin, M. Alkhalaf, and T. Bultan, *Automated test generation from vulnerability signatures*, in *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, (Washington, DC, USA), pp. 193–202, IEEE Computer Society, 2014.

[7] Y. Minamide, *Static approximation of dynamically generated web pages*, in *Proceedings of the 14th International Conference on World Wide Web*, WWW '05, (New York, NY, USA), pp. 432–441, ACM, 2005.

[8] C. Gould, Z. Su, and P. Devanbu, *Static checking of dynamically generated queries in database applications*, in *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, (Washington, DC, USA), pp. 645–654, IEEE Computer Society, 2004.

[9] S. H. Jensen, P. A. Jonsson, and A. Møller, *Remedying the eval that men do*, in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, (New York, NY, USA), pp. 34–44, ACM, 2012.

[10] H. V. Nguyen, C. Kästner, and T. N. Nguyen, *Building call graphs for embedded client-side code in dynamic web applications*, in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, (New York, NY, USA), pp. 518–529, ACM, 2014.

[11] H. V. Nguyen, C. Kästner, and T. N. Nguyen, *Varis: Ide support for embedded client code in php web applications*, in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, (Piscataway, NJ, USA), pp. 693–696, IEEE Press, 2015.

[12] F. Yu, M. Alkhalaf, and T. Bultan, *Patching vulnerabilities with sanitization synthesis*, in *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, (New York, NY, USA), pp. 251–260, ACM, 2011.

[13] M. Alkhalaf, A. Aydin, and T. Bultan, *Semantic differential repair for input validation and sanitization*, in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, (New York, NY, USA), pp. 225–236, ACM, 2014.

[14] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, *Hampi: A solver for string constraints*, in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, (New York, NY, USA), pp. 105–116, ACM, 2009.

[15] A. S. Christensen, A. Møller, and M. I. Schwartzbach, *Precise analysis of string expressions*, in *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, (Berlin, Heidelberg), pp. 1–18, Springer-Verlag, 2003.

[16] P. Hooimeijer and W. Weimer, *Strsolve: solving string constraints lazily*, *Automated Software Engineering* **19** (2012), no. 4 531–559.

[17] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner, *Symbolic finite state transducers: Algorithms and applications*, in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, (New York, NY, USA), pp. 137–150, ACM, 2012.

[18] L. D'antoni and M. Veanes, *Extended symbolic finite automata and transducers*, *Form. Methods Syst. Des.* **47** (Aug., 2015) 93–119.

[19] M. Veanes, T. Mytkowicz, D. Molnar, and B. Livshits, *Data-parallel string-manipulating programs*, in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, (New York, NY, USA), pp. 139–152, ACM, 2015.

[20] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, *A symbolic execution framework for javascript*, in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, (Washington, DC, USA), pp. 513–528, IEEE Computer Society, 2010.

[21] M. Veanes, P. d. Halleux, and N. Tillmann, *Rex: Symbolic regular expression explorer*, in *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, (Washington, DC, USA), pp. 498–507, IEEE Computer Society, 2010.

[22] G. Li and I. Ghosh, *Hardware and Software: Verification and Testing: 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings*, ch. PASS: String Solving with Parameterized Array and Interval Automaton, pp. 15–31. Springer International Publishing, Cham, 2013.

[23] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, ch. CVC4, pp. 171–177. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[24] P. A. Abdulla, M. F. Atig, Y.-F. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman, *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, ch. Norn: An SMT Solver for String Constraints, pp. 462–469. Springer International Publishing, Cham, 2015.

[25] A. Aydin, L. Bang, and T. Bultan, *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, ch. Automata-Based Model Counting for String Constraints, pp. 255–272. Springer International Publishing, Cham, 2015.

[26] N. Bjørner, N. Tillmann, and A. Voronkov, *Path feasibility analysis for string-manipulating programs*, in *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, TACAS '09, (Berlin, Heidelberg), pp. 307–321, Springer-Verlag, 2009.

[27] F. Yu, T. Bultan, and B. Hardekopf, *String abstractions for string verification*, in *Proceedings of the 18th International SPIN Conference on Model Checking Software*, (Berlin, Heidelberg), pp. 20–37, Springer-Verlag, 2011.

[28] F. Yu, T. Bultan, and O. H. Ibarra, *Relational string verification using multi-track automata*, in *Proceedings of the 15th International Conference on Implementation and Application of Automata*, CIAA'10, (Berlin, Heidelberg), pp. 290–299, Springer-Verlag, 2011.

[29] F. Yu, M. Alkhalaf, T. Bultan, and O. H. Ibarra, *Automata-based symbolic string analysis for vulnerability detection*, *Formal Methods in System Design* **44** (2014), no. 1 44–70.

[30] M. Alkhalaf, S. R. Choudhary, M. Fazzini, T. Bultan, A. Orso, and C. Kruegel, *Viewpoints: Differential string analysis for discovering client- and server-side input validation inconsistencies*, in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, (New York, NY, USA), pp. 56–66, ACM, 2012.

[31] D. Clark, S. Hunt, and P. Malacaria, *A static analysis for quantifying information flow in a simple imperative language*, *J. Comput. Secur.* **15** (Aug., 2007) 321–371.

[32] S. McCamant and M. D. Ernst, *Quantitative information flow as network flow capacity*, in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, (New York, NY, USA), pp. 193–205, ACM, 2008.

[33] G. Smith, *On the foundations of quantitative information flow*, in *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, FOSSACS '09, (Berlin, Heidelberg), pp. 288–302, Springer-Verlag, 2009.

[34] Q.-S. Phan, P. Malacaria, O. Tkachuk, and C. S. Păsăreanu, *Symbolic quantitative information flow*, *SIGSOFT Softw. Eng. Notes* **37** (Nov., 2012) 1–5.

[35] A. Filieri, C. S. Păsăreanu, and W. Visser, *Reliability analysis in symbolic pathfinder*, in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, (Piscataway, NJ, USA), pp. 622–631, IEEE Press, 2013.

[36] M. Borges, A. Filieri, M. d'Amorim, C. S. Păsăreanu, and W. Visser, *Compositional solution space quantification for probabilistic software analysis*, in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, (New York, NY, USA), pp. 123–132, ACM, 2014.

[37] C. Bartzis and T. Bultan, *Computer Aided Verification: 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004. Proceedings*, ch. Widening Arithmetic Automata, pp. 321–333. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[38] J. C. King, *Symbolic execution and program testing*, *Commun. ACM* **19** (July, 1976) 385–394.

[39] G. Redelinghuys, W. Visser, and J. Geldenhuys, *Symbolic execution of programs with strings*, in *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*, SAICSIT '12, (New York, NY, USA), pp. 139–148, ACM, 2012.

[40] Y. Zheng, X. Zhang, and V. Ganesh, *Z3-str: A z3-based string solver for web application analysis*, in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, (New York, NY, USA), pp. 114–124, ACM, 2013.

[41] M.-T. Trinh, D.-H. Chu, and J. Jaffar, *S3: A symbolic string solver for vulnerability detection in web applications*, in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, (New York, NY, USA), pp. 1232–1243, ACM, 2014.

[42] L. Luu, S. Shinde, P. Saxena, and B. Demsky, *A model counter for constraints over unbounded strings*, in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, p. 57, 2014.

[43] F. Yu, T. Bultan, and O. H. Ibarra, *Relational string verification using multi-track automata*, *Int. J. Found. Comput. Sci.* **22** (2011), no. 8 1909–1924.

[44] C. Bartzis and T. Bultan, *Efficient symbolic representations for arithmetic constraints in verification*, *Int. J. Found. Comput. Sci.* **14** (2003), no. 4 605–624.

[45] N. Biggs, *Algebraic Graph Theory*. Cambridge Mathematical Library. Cambridge University Press, 1993.

[46] P. Flajolet and R. Sedgewick, *Analytic Combinatorics*. Cambridge University Press, New York, NY, USA, 1 ed., 2009.

[47] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd ed., 2001.

[48] J. L. Gross, J. Yellen, and P. Zhang, *Handbook of Graph Theory, Second Edition*. Chapman & Hall/CRC, 2nd ed., 2013.

[49] R. P. Stanley, *Enumerative Combinatorics: Volume 1*. Cambridge University Press, New York, NY, USA, 2nd ed., 2011.

[50] B. Ravikumar and G. Eisman, *Weak minimization of DFA - an algorithm and applications*, *Theor. Comput. Sci.* **328** (2004), no. 1-2 113–133.

[51] D. E. Knuth, *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1968.

[52] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters, *A DPLL(T) theory solver for a theory of strings and regular expressions*, in *Proceedings of the 26th International Conference on Computer Aided Verification (CAV)*, pp. 646–662, 2014.

[53] S. Kausler and E. Sherman, *Evaluation of string constraint solvers in the context of symbolic execution*, in *Proceedings of the 29th ACM/IEEE International Conference on Automated software engineering (ASE)*, pp. 259–270, 2014.

[54] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, *A symbolic execution framework for javascript*, in *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.

[55] F. Yu, M. Alkhalaf, and T. Bultan, *Stranger: An automata-based string analysis tool for php*, in *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 154–157, 2010.

[56] V. Baldoni, N. Berline, J. D. Loera, B. Dutra, M. Köppe, S. Moreinis, G. Pinto, M. Vergne, and J. Wu, "Latte integrale v1.7.2." http://www.math.ucdavis.edu/ latte/.

[57] J. A. D. Loera, R. Hemmecke, J. Tauzer, and R. Yoshida, *Effective lattice point counting in rational convex polytopes*, *Journal of Symbolic Computation* **38** (2004), no. 4 1273 – 1302.

[58] A. Aydin, L. Bang, and T. Bultan, *Automata-based model counting for string constraints*, in *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, Proceedings, Part I*, pp. 255–272, 2015.

[59] A. Filieri, C. S. Pasareanu, and W. Visser, *Reliability analysis in symbolic pathfinder*, in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pp. 622–631, 2013.

[60] L. Bang, A. Aydin, Q.-S. Phan, C. S. Păsăreanu, and T. Bultan, *String Analysis for Side Channels with Segmented Oracles*, in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, (New York, NY, USA), ACM, 2016.

[61] D. Balasubramanian, K. Luckow, C. Pasareanu, A. Aydin, L. Bang, T. Bultan, M. Gavrilov, T. Kahsai, R. Kersten, D. Kostyuchenko, Q.-S. Phan, Z. Zhang, and G. Karsai, *ISSTAC: Integrated Symbolic Execution for Space-Time Analysis of Code*, in *submission*, 2017.

[62] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott, *The omega calculator and library, version 1.1. 0, College Park, MD* **20742** (1996) 18.

[63] F. Yu, M. Alkhalaf, and T. Bultan, *Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses*, in *ASE*, 2009.

[64] R. E. Tarjan, *Depth-first search and linear graph algorithms*, *SIAM J. Comput.* **1** (1972), no. 2 146–160.

[65] M. Brandizi, N. Kurbatova, U. Sarkans, and P. Rocca-Serra, *graph2tab, a library to convert experimental workflow graphs into tabular formats*, *Bioinformatics* **28** (2012), no. 12 1665–1667.

[66] S. C. Ntafos and S. L. Hakimi, *On path cover problems in digraphs and applications to program testing*, *IEEE Trans. Software Eng.* **5** (1979), no. 5 520–529.

[67] E. Ciurea and L. Ciupal, *Sequential and parallel algorithms for minimum flows*, *Journal of Applied Mathematics and Computing* **15** (2004), no. 1-2 53–75.

[68] L. Ford Jr and D. Fulkerson, *Maximal flow through a network*, in *Classic papers in combinatorics*, pp. 243–248. Springer, 1987.

[69] F. Yu, M. Alkhalaf, and T. Bultan, *Stranger: An automata-based string analysis tool for php*, in *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 154–157, 2010.