Symbolic String Execution

by

Gideon Redelinghuys

Thesis presented in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science

at the University of Stellenbosch

Department of Computer Science,
University of Stellenbosch,
Private Bag X1, 7602 Matieland, South Africa.

Supervisors:

Prof W. Visser   Dr. J. Geldenhuys

2012

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the owner of the copyright thereof (unless to the extent explicitly otherwise stated) and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Signature: Gideon Redelinghuys ....

G. Redelinghuys

Date: 1 October 2011 ....

Abstract

Symbolic String Execution

G. Redelinghuys

Department of Computer Science,

University of Stellenbosch,

Private Bag X1, 7602 Matieland, South Africa.

Thesis: MSc

2012

Symbolic execution is a well-established technique for automated test generation and for finding errors in complex code. Most of the focus has however been on programs that manipulate integers, booleans, and even, references in object-oriented programs. Recently researchers have started looking at programs that do lots of string processing, motivated, in part, by the popularity of the web and the risk that errors in web servers may lead to security violations. Attempts to extend symbolic execution to the domain of strings are mainly divided into one of two camps: automata-based approaches and approaches based on bitvector analysis. Here we investigate these two approaches in a unified setting, namely the symbolic execution framework of Java PathFinder. We describe the implementations of both approaches and then do an evaluation to show under what circumstances each approach performs well (or not so well). We also illustrate the usefulness of the symbolic execution of strings by finding errors in real-world examples.

Uittreksel

Simboliese Uitvoering van Stringe

("Symbolic String Execution")

G. Redelinghuys

Departement Rekenaarwetenskap,

Universiteit van Stellenbosch,

Privaatsak X1, 7602 Matieland, Suid Afrika.

Tesis: MSc

2012

Simboliese uitvoering is 'n bekende tegniek vir automatiese genereering van toetse en om foute te vind in ingewikkelde bronkode. Die fokus sover was grotendeels op programme wat gebruik maak van heelgetalle, boolse waardes en selfs verwysings in objek geörienteerde programme. Navorsers het onlangs begin kyk na programme wat baie gebruik maak van string prosessering, deelteliks gemotiveerd deur die populariteit van die web en die gepaardgaande risiko's daarvan. Vorige implementasies van simboliese string uitvoering word binne twee kampe verdeel: die automata gebaseerde benadering en bitvektoor gebaseerde benadering. Binne hierdie tesis word die twee benaderings onder een dak gebring, naamliks Java PathFinder. Die implentasie van beide benaderings word bespreek en ge-evalueer om die omstandighede uit te wys waarbinne elk beter sou vaar. Die nut van simboliese string uitvoering word geïllustreer deur dit toe te pas in foutiewe regte wêreld voorbeelde.

Contents

Chapter 1

Introduction

Adequate testing of software is hard and expensive [22]. Furthermore, attempting to achieve this by manually creating a set of tests is not only hard but also unmaintainable. Therefore, techniques which provide an automated investigation and testing of software, is not only a desired route, but a necessity.

In the past, random generation of input, or "fuzzing" of user provided input [11, 16], has produced some interesting results. Unfortunately some behaviours are relatively scarce and can only be triggered by a few inputs. In these cases a random approach is unlikely to hit upon the appropriate inputs, and user-supplied input may not help either. A more powerful approach is symbolic execution [19] which is capable of reasoning about the behaviour of the program and generating input to invoke it. Symbolic execution is a white-box technique which allows a test generator to partition the possible behaviours of a given program by determining all possible branches that could be taken during the execution. Each partition is represented by a *path condition.* Every path condition is checked for satisfiability, and if satisfiable, determines explicit input values that can be used to reproduce the associated behaviour of the software. The set of partitions may be infinite but established techniques can be applied to produce a feasible finite subset.

Symbolic execution has been applied to programs that manipulate real numbers and object references [17, 24]. Recently, programs that manipulate strings have received new interest because of the realisation that symbolic execution over symbolic strings can identify security vulnerabilities within software [7]. Our work is concerned with executing Java code (uninstrumented) and applying symbolic execution to symbolic strings and integers. The goal is not only test generation, but also checking whether given behaviours (such as those that lead to an error, or inconsistent state) are feasible.

Why is testing of string manipulating programs important? Many applications rely heavily on text processing, but the growing use of the Internet for interactive applications (such as social networks, information and entertainment services, managing sensitive, sometimes personal information) has made this problem more acute. Text inputs are often used in an SQL query and passed on the service's database [13]. Unfortunately this allows the user direct access to the database and, because these text inputs are open to the public, it is also open to wide audience, some of whom have malicious intentions. For this reason text input sanitisation is now found in almost all web services to prevent users from abusing the service. In Section 4.1 we give an example where input needs to be sanitised by stripping some characters from the input string to make sure no malicious actions can result. In one real-world application (which we are not at liberty to discuss) the input "`<< HREF=""<A HREF=">        `" caused an infinite loop in the system, resulting in a lengthy service outage. In this example, the fact that potentially malicious input was sanitised actually caused an error (even though the input was not malicious).

Why is symbolic execution of string manipulation hard? String operations mix two domains, namely strings and integers. One example of this is an operations that retrieves the n-th character of a string. Many of the current solutions to symbolic execution for strings support only a subset of such operations or often none at all. We take an iterative approach where we first solve the integer constraints and then use the results to solve the string constraints. If they are satisfiable, we are done. Otherwise, additional integer constraints are generated and the process is repeated.

How are we doing the symbolic analysis? Existing approaches to symbolic execution of string code can be divided into two groups: automata-based [4, 14, 15, 29, 28] and bitvector-based [3, 18, 26, 32]. One of our main contributions is a comparison of these two approaches within one setting. The setting we choose is that of symbolic execution of Java programs, and specifically the symbolic execution extension of the Java PathFinder (JPF) model checker [33]. The symbolic execution extension of JPF (called *JPF-symbc*, or, Symbolic PathFinder) supports symbolic analysis of many domains, including real numbers and object references. There is also a proprietary implementation for string analysis used by Fujitsu based on the automata approach [28]. *JPF-symbc* supports a wide variety of decision procedures to handle the non-string domains, and our solution for strings is engineered in such a way that it can be used in combination with any of these, with one important caveat: We can only use those decision procedures that have the capability to provide satisfying solutions, i.e., solve constraints. For this reason we refer to the decision procedures for the integer domain as constraint solvers in the rest of the thesis. We use the automata package of the Java String Analyzer (JSA) [4] for our automata approach and the Z3 SMT solver [5] for bitvectors. In both cases these solutions are also used in other string symbolic execution engines: Fujitsu and JSA itself uses the automata package from JSA, and PEX uses Z3.

How much can we solve before using these tools? For every string constraint that our tool encounters during the analysis we first build a constraint graph, called a *string graph*. Using some straightforward heuristics we then simplify the graph and if possible find inconsistencies that immediately show the unsatisfiability of the constraints. The string graph can be seen as an intermediate representation, since after simplification it is translated into the back-end format required by either the automata- or bitvector approach.

What did we find? From an implementation point of view there is a considerable difference between the two approaches: one needs to build a string decision procedure on top of the automata package, whereas the SMT solver has many of the required functionality already built in. After translation of the string graph into bitvectors, it is essentially push-button. In order to evaluate the relative performance we did a number of experiments on both artificially generated and real-world examples. Our technique found the error mentioned above in a few minutes and detected the error described below in Section 1.1

(that formed the basis of an actual security attack) in a few seconds. Interestingly we found that, on the whole, automata- and bitvector-based back-ends perform similarly, but that the real important part of the system is how one handles the interaction between string and integer constraints.

The contributions of this work can be summarised as follows:

- The introduction of the string graph data structure for representing constraints along with preprocessing heuristics.
- A detailed description of how mixed integer and string constraints are handled.
- A detailed and novel comparison of automata- and bitvector-based back-ends for string symbolic execution.
- An evaluation on both artificial examples (to determine the strengths and weaknesses of each approach) and real-world programs to show the effectiveness of the tool.
- An open source extension of *JPF-symbc*[1], including all the examples found in this thesis.

The rest of this chapter provides a more detailed motivation for the work that follows. Chapter 2 deals with an overview of the background knowledge used to research this work, Chapter 3 outlines our approach, with discussions of our findings, and Chapter 4 applies our work to artificial and real-world examples.

1.1 Motivation

It is rare to find a sizeable software package complete devoid of string operations. String sanitisation in particular is frequently used to clean input and remove any malicious content. Without this, a user could manipulate the software in an undesirable fashion. For example, many websites accept input data, transform and send it to some (typically relational) database via the

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

SQL language. If the input is not sanitised, the user could craft SQL queries that are executed by the database, causing the database to reveal or modify sensitive data.

Security is not the only concern. The Java string library is vast with room for many mistakes. We are also concerned with identifying bugs, so that our work could be applied to string intensive software to give better coverage during testing, and lead to a more stable product.

Consider function `site_exec` in Figure 1.1. It is part of the `wu_ftpd` implementation of the file transfer protocol (FTP), ported from C to Java. Its purpose is to receive and execute remote commands. If the command extracted from the input contains the substring "`%n`", a runtime exception is thrown (in line 17). Although this situation is harmless in Java, in the original C implementation it could potentially allow the user to alter the program stack and to take control of the FTP server. Detecting this kind of code injection is one of the important applications of symbolic string execution, and this example, although somewhat artificial, illustrates a typical scenario. This example is taken from a real application and is based on a real error.

One possible input string *cmd* that will trigger the runtime exception is one which satisfies the following constraints (s_2 and i are auxiliary variables):

$$\begin{aligned}
& \mathit{cmd}.\texttt{indexOf}(\text{`␣'}) = -1 \\
\wedge\quad & \mathit{cmd}.\texttt{lastIndexOf}(\text{`/'}) \geq 0 \\
\wedge\quad & \mathit{cmd}.\texttt{lastIndexOf}(\text{`/'}) = i \\
\wedge\quad & \mathit{cmd}.\texttt{substring}(i) = s_2 \\
\wedge\quad & s_2.\texttt{length}() < 19 \\
\wedge\quad & s_2.\texttt{contains}(\text{``}\texttt{\%n}\text{''})
\end{aligned}$$

We refer to the last constraint as a (pure) string constraint, because it involves only string variables and constants. The second last constraint, on the other hand, is a (pure) integer constraint, since s_2.`length` is in essence an integer variable and 19 is an integer constant. The other constraints are mixed (integer and string) constraints.

For this work we will only consider faulty behaviour that leads to an explicit exception being thrown (such as the on on line 17). Any implicit faults

```
 1  public void site_exec(String cmd) {
 2     String result;
 3     String path = "/home/ftp/bin";
 4     int j, sp = cmd.indexOf(' ');
 5     if (sp == -1) {
 6        j = cmd.lastIndexOf('/');
 7        result = cmd.substring(j);
 8     } else {
 9        j = cmd.lastIndexOf('/', sp);
10        result = cmd.substring(j);
11     }
12     if (result.length() + path.length() > 32) {
13        return; // buffer overflow
14     }
15     String buf = path + result;
16     if (buf.contains("%n")) {
17        throw new Exception("THREAT");
18     }
19     execute(buf);
20  }
```

Figure 1.1: Example of code injection

resulting from abnormal use of the String API is not considered. An example of such an implicit fault is if j, in the given example, is equal to -1 at line 7 or line 10.

This classification is clearly important, because different decision procedures and constraint solvers are required for different kinds of constraints. Chapter 3 describes the details of how the constraints are represented, how and when information is passed between the integer and string solvers, and how string and mixed constraints are handled by automata and bitvector constraint solvers.

1.2 String use in software

The Java language provides a number of `String` operations. Like all other tools, the tool developed in this work only executes a subset of these operations in a sound and complete manner. JSA approximates the entire Java String

Returns	Operation	Notes
`boolean`	`b.startsWith (String a)`	Returns true if `b` starts with `a`
`boolean`	`b.endsWith (String a)`	Returns true if `b` ends with `a`
`boolean`	`b.equals (String a)`	Returns true if `b` has the same length and the same sequence of characters as `a`.
`boolean`	`b.contains (String a)`	Returns true if `a` is contained within `b`
`String`	`b.trim()`	Returns the string that results from removing leading and trailing whitespaces from `b`
`String`	`b.concat(String a)`	Returns the string that results from appending `a` to the back of `b`. Tends to be the most difficult constraint some of the other string constraint solvers attempt to solve.
`String`	`b.substring(int i)`	Returns the string that starts from index `i` of `b`. Support for `substring` in other solvers tend to have `i` as an integer constant.
`String`	`b.substring(int i, int j)`	Returns the string that starts from index `i` and ends at index `j - 1` of `b`
`int`	`b.length()`	Returns the number of characters in `b`
`char`	`b.charAt(int i)`	Returns the character at index `i` in `b`. Other solvers force `i` to be constant.
`int`	`b.indexOf(char c)`	Returns the index of the first occurrence of `c` in `b`
`int`	`b.indexOf(char c, int i)`	Returns the index of the first occurrence of `c`, after index $i - 1$, in `b`
`int`	`b.indexOf(String s)`	Returns the index of the first occurrence of `s` in `b`
`int`	`b.indexOf(String s, int i)`	Returns the index of the first occurrence of `s`, after index $i - 1$, in `b`

Figure 1.2: What is considered 'common' string operations

API. Figure 1.2 is a list of the "common" string operations in Java with notes and is worth studying because we will be using Java programs as examples.

Implementing support for the entire Java `String` API is quite a feat, and is not attempted in this work. Rather, operations were prioritised and support was added as needed. We found that extending our approach was easy, an important fact, since some solvers lack the capabilities to solve some categories of string operations without major rethinking and reengineering.

Prioritising operations was achieved by inspecting a sample of projects, large and small, that are freely available on the Internet. With a Python script string operations from the `java.lang.String`, `java.lang.StringBuilder` and `java.lang.StringBuffer` libraries were counted. The script counted operations by looking at each project's Java Virtual Machine byte code.

At first only operations from the `java.lang.String` class were counted, but this led to an incorrect conclusion. Java programs seem to have plenty of concatenation of strings, and the Python script did not pick it up. Only after further investigation did it occur to us that programmers that develop large public projects apply certain techniques to achieve better performance, including concatenating strings as fast as possible with the use of the methods available in the `java.lang.StringBuilder` and `java.lang.StringBuffer` classes. For example, a programmer who is not aware of the subtle performance bottlenecks in the Java library may produce the code shown in Figure 1.3(a). A more experienced programmer would rewrite it as in Figure 1.3(b). The example in Figre 1.3 is for demonstration purposes, it only provide a speedup if more than two string variables are involved. Our inspection of a wide sample of popular open source Java projects shows that the Java programmers working on all of these projects also use the latter form of concatenation. Of all the String operations counted, fewer then 20 were `java.lang.String.concat`, whereas the use of the append methods in `java.lang.StringBuilder` and `java.lang.StringBuffer` was in the hundreds of thousands. For the rest of this work we will refer to both concatenation and appending as concatenation.

A problem with our counting of string operations is that it is a static inspection whereas symbolic execution is dynamic. Thus the inspection might report, for example, that Project A uses `equal` only once, while `equal` might actually be used many times during execution (in a loop for instance). Gathering string operation data dynamically is difficult due to scalability problems, lack of domain knowledge for each project and because it is a function of the input distribution. Given this limitation, we still believe that inspecting the code statically leads to a good estimation of the importance of each string

```
public static String concatSlow (String a, String b) {
        return a + b; //or a.concat(b)
}
```

(a) Concatenating Strings slowly

```
public static String concatFast (String a, String b) {
        StringBuffer sb = new StringBuffer(a);
        return a.append(b).toString();
}
```

(b) Concatenating Strings quickly

Figure 1.3: Slow vs fast concatenation

operation.

It is natural to expect that the popularity of each String operation varies with the nature of the project. In order to verify this assumption a broad range of applications was selected by hand. The selected Java projects are given in Figure 1.4. Descriptions of the projects have been added so that the reader can verify that these projects come from a diverse background.

The most startling result is the use of concatenation. Figure 1.5 compares the three most popular operations with the rest. Concatenation was found to account for almost 70% of the operations, while `equals` and `length` were, approximately, 10% and 4% respectively. The methods `toString` and `format` were ignored because they do not imply any explicit constraints upon string variables. Figure 1.6 gives more information on the 11 most popular operations (excluding any concatenation operation). Interestingly, there is a focus by researchers on solving `replace` effectively [6], while we found it to be less then one percent of operations. Operations such as `length`, `substring`, `indexOf` and `charAt` all need accurate symbolic string-integer constraint solving because it is clear that they are popular among the programs inspected. By this metric, concatenation are clearly the "most important" operations, but we choose to ignore them in our string operation counting, because we feel that defects in string handling normally do not occur in concatenation. The string constraint solver is still required to support concatenation.

The expectation that in different projects the popularity of `String` operations would differ was found to be untrue. Almost all projects conformed

Project	**Description**
Ant	Automated build tool for Java
ANTLR	Another Tool for Language Recognition
Apache Camel	Provides an object-orientated API to implement rule-based routing and mediation rules
Apache Commons DBCP	Database connection pool
Apache Commons Validator	Data validation
Apache CXF	Web services framework
Apache Derby	Relational database implemented in Java
Apache Jackrabbit	Open source content repository
AspectWerkz	Aspect-oriented programming (AOP) framework
Checkstyle	A tool to help programmers write source code that adheres to a coding standard
Coefficient	Collaboration tool for work environment
DjVu	Viewer of scanned documents that are stored in the DjVu format
DrJava	Lightweight IDE for writing Java programs
Drools	Object-oriented rule engine
DSpace	Digital library system that manages the intellectual output of researchers
FindBugs	Uses static analysis to look for bugs in Java code
Google Web Kit	Development framework for AJAX applications
Heritrix	Web crawler project
Hibernate	Relational persistence for Idiomatic Java
HtmlUnit	Unit testing framework for testing web based applications
JabRef	Java based LaTeX BibTeX manager
JArgs	Command line option parsing suite
JBoss	Java EE-based application server
JEdit	Text editor for programmers
JFreeChart	Library for generating charts
JMoney	Personal finance manager
JSPWiki	WikiWiki web clone
JUnit	Testing framework
LlamaChat	Chat server/client pair for use on the web
Log4j	Logging tool
Paros	HTTP/HTTPS proxy for assessing web application vulnerability
PMD	Scans Java source code and looks for potential problems
ProGuard	Java class file shrinker and obfuscater
Report design	Eclipse plugin that makes it easier to create a report file
RES	Open Cobol to Java Translator
SQuirreL SQL Client	Graphical SQL Client for JDBC
TagSoup	Parser for HTML “as it is found in the wild”
Tapestry	Framework for creating web applications

Figure 1.4: Java projects used to gather string operation usage

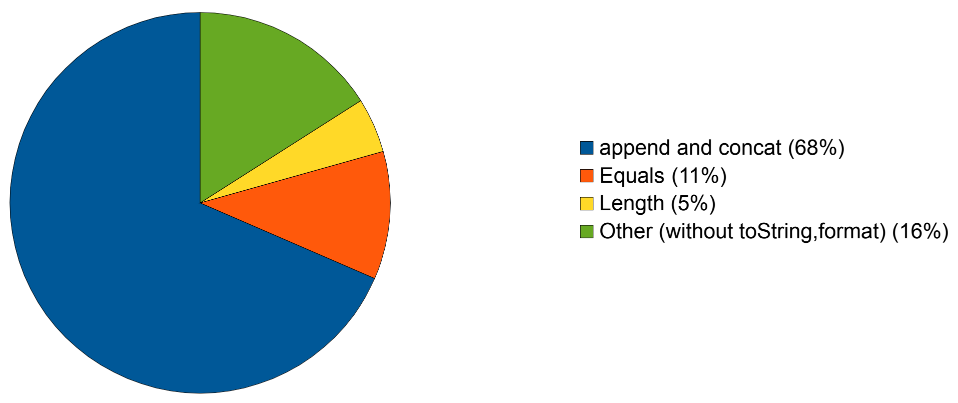


Figure 1.5: Append and concat vs. other Java String operations

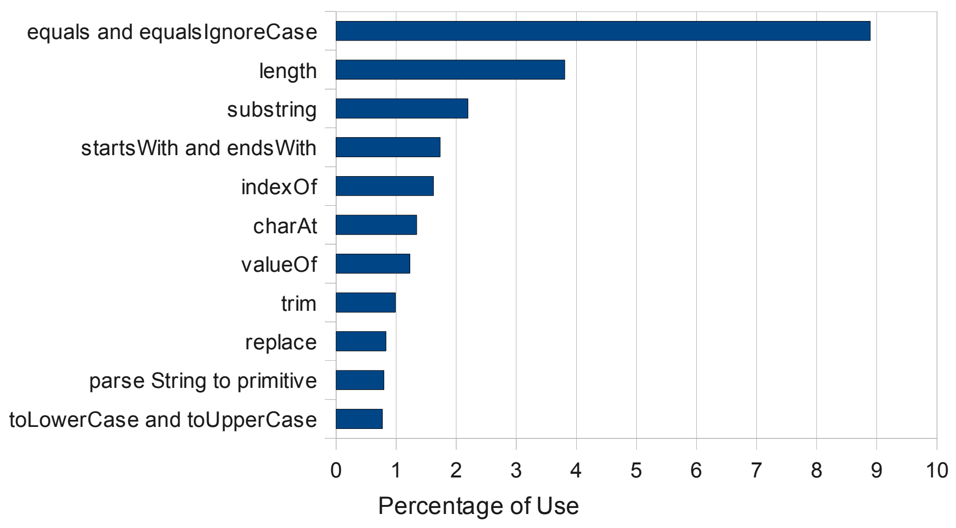


Figure 1.6: String operation usage in Java projects

individually to Figure 1.5. Conformance to Figure 1.6 differed slightly with some operations varying by a ranking or two.

As far as we are aware, there are only two published studies mentioning the frequency of string operations. Saxena [26] found that in JavaScript programs, `indexOf` and `length` accounted for 78% of operations while `concat` made up 8%, `replace` 8%, `substring` and `charAt` 5%, and `split` 1%. JSA [4] found `concat` to be their "most important string operation" in Java programs (which is reflected in our results as well).

To create the most effective solver with the minimal string operation support we determined the smallest possible subset of operations to support. From all 38 programs, all used a form of: {`equals`, `indexOf`, `length`, `substring`} at least once. If a string constraint solver wishes to support the run of any entire program, these four operations must be supported. We will label these four operations as the four *base* operations. The four base operations alone are not enough to run any one of the example programs. Some of the operations in the list below will also be required:

`capacity`	`endsWith`	`parseFloat`	`startsWith`
`charAt`	`equalsIgnoreCase`	`parseInt`	`subSequence`
`compareTo`	`intern`	`regionMatches`	`toCharArray`
`concat`	`isEmpty`	`replace`	`toLowerCase`
`contains`	`lastIndexOf`	`setCharAt`	`toUpperCase`
`contentEquals`	`matches`	`setLength`	`trim`
`copyValueOf`	`parseDouble`	`split`	`valueOf`

Adding support for `charAt` and `startsWith` operations would make a string constraint solver capable of solving all constraints encountered in the JArgs project. If a string constraint solver wishes to be able to solve at least two projects it should at least be able to support `charAt`, `lastIndexOf`, `parseInt`, `startsWith` and `valueOf` (making it capable of solving JUnit and JArgs). Figure 1.7 shows that as the number of supported operations increase, the number of projects supported increases significantly.

Figure 1.7 was created by calculating the maximum number of projects supported if x operations were selected from the original 28 operations. E.g.,

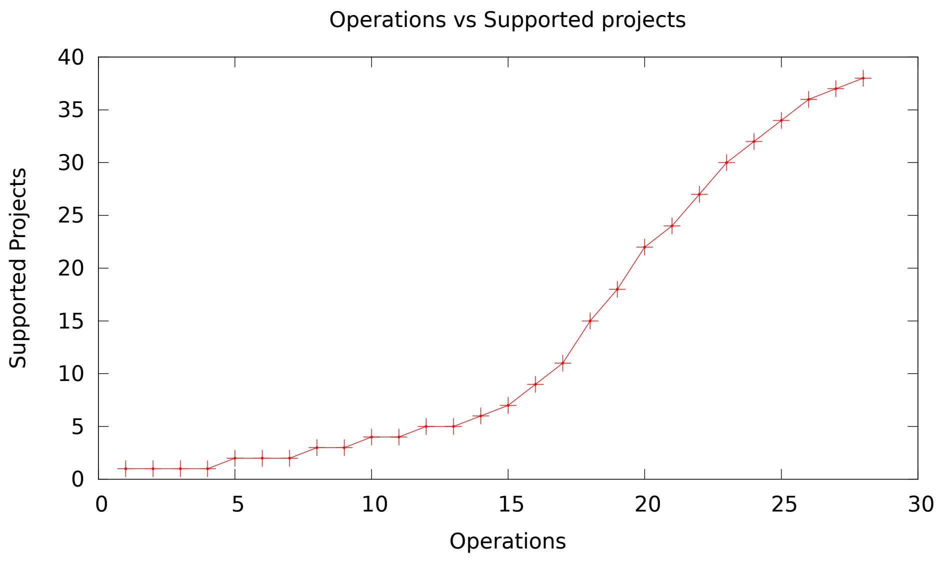


Figure 1.7: Supported projects increase in a dramatic fashion as supported operations increase.

if $x = 2$ all subsets X of cardinality 2 will be selected from the given 28 operations. The number of projects supported for a given subset in X is calculated. The subset delivering the most supported projects is used to determine the most supported projects for that cardinality.

1.3 Overview

This section described the problems facing testing, and a technique to combat those problems. The technique proposed is *symbolic execution*. It provides a method of reasoning about the input of a program to generate tests. In the industry there are software problems that are caused by incorrect or faulty string handling which string symbolic execution would be able to catch. Unfortunately, there are many string operations and so they must be prioritised to be able to create the most effective string symbolic execution technique.

Given that string symbolic execution is a valuable technique to develop, we first need to cover how it will be applied to software. For this, the next section will describe the environment (JPF) and the basic building blocks for achieving this (automata and bitvectors).

Chapter 2

Background

Our work is built upon ideas and developments that have been shaped over many years. Three key technologies we use are:

- Symbolic Execution (covered in Section 2.1 and Section 2.2)
- Automata theory (covered in Section 2.3)
- Bitvectors and SMT-solvers (covered in Section 2.4)

This section describes the basis of the entire thesis, and environment used to execute it in. The basic building blocks of implementing such a technique is also discussed. Finally, we give an overview of how other published work has used these technologies.

2.1 Symbolic Execution

Testing the entire domain of a program is impractical. An ideal method is to simply feed every possible element of the domain into the program and to verify that it satisfies all intended properties. Unfortunately, the domain may be infinite. Even if the domain can be described by a finite set of elements, it may be so large that it is infeasible to execute all of them.

When an element of the domain is used as input to a program, that element causes a set of program states to be exercised. To demonstrate this, imagine a program simulating an automatic sliding door. The program has two inputs, whether the door is closed or open, and the number of people near the door. If the door is open and there is no people detected it will exercise the states: *closing*, *closed*. The set of possible states that can exist during a program's execution is dependant on the environment executing the program.

In most cases the domain can be partitioned into equivalence classes:

> *Let any two elements of the domain of a program be equal when both exercise the exact same ordered set of states.*

If the number of equivalence classes is significantly less than the number of elements in the domain, then the domain may be more tractable.

Unfortunately, if the program has an infinite domain and an infinite set of states, it may also have an infinite number of equivalence classes. One practical solution is to limit the set of states and the domain of inputs. This is not ideal but in practice it works well.

In our automatic sliding door example the number of states is finite, but our domain is infinite. If one or more people is detected, the program will exercise the exact same ordered set. If no person is detected, a different ordered set of states is exercised. Thus, we have two equivalence classes, one for the case if there is one or more people and the other if there is no person.

A popular method of partitioning the input domain into equivalence classes is *symbolic execution*. It uses *path conditions* to represent each class. A path condition is a conjunction of boolean constraints which are satisfied. Each boolean constraint represents a branch of the execution tree that the program has taken when it reached a decision point (such as an if-statement)

Constructing the path conditions occurs during the dynamic execution of the program. When the dynamic execution starts there is only one path condition with the value:

$$\text{PC}(0): \quad \textit{true}$$

The program's instructions are then observed one by one, and when the first branch condition q_0 is found, the path condition is split into two. The first is appended with the constraint q_0 and the other with the constraint $\neg q_0$.

$$\begin{array}{ll} \text{PC}(0.0): & \mathit{true} \wedge \\ & q_0 \end{array}$$

$$\begin{array}{ll} \text{PC}(0.1): & \mathit{true} \wedge \\ & \neg q_0 \end{array}$$

The path condition PC(0.0) is then used to follow the *true* branch and the process is repeated. Once the end of that branch is reached, path condition PC(0.1) is used and the *false* branch is followed. In other words, the execution of the program is explored in a depth-first fashion.

Once all path conditions have been constructed, each one needs to be solved to obtain at least one set of values *representatives* of their respective path conditions. These representatives are used as examples to reproduce the program's execution.

The original use of symbolic execution applied to integers and real numbers [19] It can be used to track software changes [23] and for dynamic symbolic execution: the symbolic execution of data structures [17].

A path condition describing an entire execution is not constructed at once, and solved once. The path condition is solved repeatedly as a new constraint is added. Any path condition that does not yet describe an entire execution is called a *partial path condition* and represents a set of equivalence classes. If a partial path condition is found to be unsatisfiable, the approach backtracks and continues building up a different path condition.

Not all variables need to be seen as symbolic. A symbolic variable is a variable which is seen as part of the input domain, a concrete variable is assumed to be fixed (or constant). Symbolic execution in the JPF tool, described in the next session, is configurable to only run certain variables as symbolic and others as concrete, although any variable that is dependent on some symbolic variable must also be symbolic. This gives symbolic execution a great advantage when scaling to larger programs. Concolic execution [16, 27] extends the scalability of this approach even further by forcing symbolic operations to be performed as if some symbolic variables are temporarily concrete (non-symbolic).

2.2 Java PathFinder

Java PathFinder (JPF) [33] is an open-source implementation of the Java Virtual Machine for verifying Java bytecode, developed at NASA. It is a tool to analyse the states of a Java program and it consists of a core package (as described below) with many extensions. We are particularly interested in the symbolic execution extensions that has been created for it: JPF-symbc.

2.2.1 JPF-core

JPF-core has a configurable search strategy to walk through the states of program. The configurable search strategy is by default a depth-first search of a program's state space. A search strategy determines the order in which states are explored, and can have great effect on performance if a certain property needs to satisfied. Other search strategies include breadth-first search and a priority-queue based search that can be parameterised to do various search types based on selecting the most interesting state out of the collection of all successors of a given state (called Heuristic Search [12] within JPF-core). The previous three search strategies are deterministic, but a non-deterministic random search strategy is also available.

For JPF-core to verify Java code, it has to have its own virtual machine implementation. The payoff for this redundancy is that the model checker has more control over intricate details such as thread scheduling and it enables extensions to inherit and extend the virtual machine. Each extension has at least the power of JPF-core.

One of the main benefits of using JPF in research is its ease of extensibility. It was specifically developed with this in mind. Some extensions available for JPF-core are UDITA [9] which derives more test cases from already defined tests, Basset [20] which is able to automatically test actor based programs, and MuTMuT [10] which is an automated mutation tester.

2.2.2 JPF-symbc

The JPF-symbc extension allows symbolic execution of Java programs that contain integer and real numbers, booleans, references and strings (the focus of this work).

JPF-symbc replaces JPF-core's operations with its own symbolic execution-aware operations. User-selected variables are replaced with symbolic variables and any other variables dependant on them are automatically declared symbolic. Variables are kept concrete as far as possible.

JPF-core still provides the search through the Java program's control flow, but it is now complemented by the JPF-symbc extension which builds path conditions, translation of path conditions and passing them onto off-the-shelf solvers.

Just like a standard JVM, the JPF-core virtual machine maintains a stack of values and corresponding attributes for each value. These attributes allow the VM to keep track of stack frames, threads, callee attributes, caller attributes, and scheduling information. In JPF-symbc additional attributes are used to keep track of symbolic variables and the expressions that are formed by symbolic operations.

One of the advantages of JPF-core is state matching, but due to the nature of symbolic execution (each state represents a path condition on unbounded data) state matching becomes undecidable. Possible infinite branches (such as loops or recursion) are bounded by JPF-symbc (by bounding JPF-core's search depth).

When executing a Java program under JPF-symbc one can specify which parameters of methods are to be treated symbolically. The fewer symbolic variables present in the symbolic execution the quicker it will terminate. Therefore, if a user knows that certain variables are not relevant to the exploration, he can mark them as concrete.

After symbolic execution has solved all path conditions, it concludes with either a set of unit tests which would, ideally, exercise all reachable code or a certain input which violates a specified property. The former will occur when no defects were found in the software, in which case JPF-symbc generates a unit test for each path condition. Each unit test simply calls the method specified by the user with a representative from that path condition. The latter will occur as soon as it is found that the program can throw an exception or fail an assertion.

JPF-symbc has been used successfully with the NASA On-board Abort Executive to identify fatal defects in the software that were fixed before they were found in the field [25].

The work outlined in this thesis now forms part of JPF-symbc by extending the numeric symbolic operations with string operations.

2.3 Automata theory

It is natural to think of automata when searching for a way to represent strings and string constraints: strings are words (over some alphabet) and string variables can store languages of words. Furthermore, one may expect a natural mapping from string operation to automata operations. If a given set of string constraints can be translated into a set of automata equations, the problem of solving them can be based on an area that has been researched exhaustively.

Definition 1 *A finite automaton is a 5-tuple:* $\{Q,\ \Sigma,\ \delta,\ q_0,\ F\}$ *[30] where*

1. Q *is a finite set called the states,*
2. Σ *is a finite set called the alphabet,*
3. δ*:* $Q \times \Sigma \rightarrow Q$ *is the transition function,*
4. $q_0 \in Q$ *is the start state, and*
5. $F \subseteq Q$ *is the set of accept states.*

An automaton takes a string of the form a_1,a_2,...,a_n where $a_i \in \Sigma$ as input. Each symbol of the input string leads to a sequence of states q_0,q_1,...,q_n where $q_i \in Q$ such that q_0 is the start state and $q_i = \delta(q_{i-1}, a_i)$ for $0 < i \leq n$. An input word is accepted if $q_n \in F$.

Automata can occur as nondeterministic finite automata (NFA) and as Deterministic Finite Automata (DFA). An NFA also allows nondeterministic decisions and has an added ϵ symbol in its alphabet, which enables it to take transitions nondeterministicly without consuming any input symbol. Both forms are equivalent, but NFA can simplify the translation from one automata to another during an operation. After all translations are applied, any NFA are converted to its equivalent DFA.

It is well-known that the languages recognised by DFA and NFA are exactly the regular languages [30] A regular language is defined (over an alphabet Σ) recursively as follows:

- The empty language $\emptyset$ is a regular language.
- The empty string language $\{\epsilon\}$ is a regular language.
- For each $a \in \Sigma$, the singleton language $\{a\}$ is a regular language.
- If A and B are regular languages, then $A \cup B$ (union), $A \oplus B$ (concatenation), and A^* (Kleene star) are regular languages.
- No other languages over Σ are regular.

Regular languages cannot describe all possible languages. More importantly, it cannot describe the language that satisfies many sets of string constraints. However, if the string variables are bounded by some length, they can be described by a finite set of words, which is again a regular language.

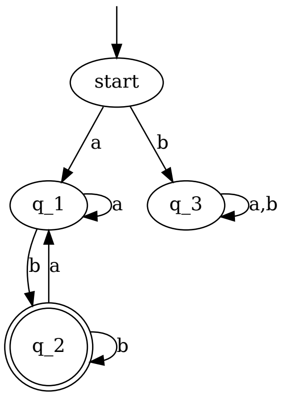
For example, consider the string constraint that some string variable must start with n open brackets and end with n closing brackets. No regular language can describe the entire set of words that will satisfy this constraint. However, if it is added that the string variable may be no longer than 4 characters, the set of words satisfying the constraint becomes finite ($\{\epsilon$, `()`, `(())`$\}$). By bounding the length of all string variables, all solutions become expressible as the regular language.

A regular expression is equivalent to a regular language. Regular expressions are a popular way of expressing regular languages. Regular expressions are often used to describe security vulnerabilities.

A regular language can be described in many ways, such as finite state machine graph and regular expressions (Figure 2.1). Regular expressions will be used in most of this work due to its compactness and frequent use in this field of research (sanitisation checking and string constraint solving).

`a.*b`

(a) Regular Expression



(b) Finite State Machine Graph

Figure 2.1: Two ways of expressing an automata

Character	Description
.	Describes any character in the language
*	The prefixed character (or set of characters) is repeated zero or more times
+	The prefixed character (or set of characters) is repeated one or more times
[s_1 - s_2]	Describes one characters in the range between (and including) s_1 and s_2

Figure 2.2: Overview of regular expressions

Figure 2.2 is given as a quick overview of the language of regular expressions.

In this thesis, the regular language operations **intersection** ($\cap$) and **concatenation** ($\oplus$) are used almost exclusively to alter the state of the regular languages used to describe string variables.

Given two automata (or equivalently two sets of words), their intersection is an automaton that accepts the set of those words that are accepted by both automata.

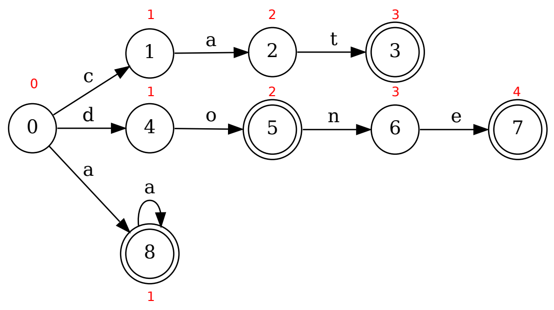


Figure 2.3: An example automaton for demonstrating the substring operation. For the sake of abbreviation we have not included that for every state there is a transition from that state to some non-accepting state for the transitions not defined.

If automaton M is intersected with another automaton and the result is stored in M, and if that process is repeated the language of M shall never grow larger.

Two regular languages can be concatenated to create a regular language where for each accepted word, the first part of that word is described by the first regular language and the rest of the word is described by the second regular language.

Throughout this work when it is stated that a regular language operation is applied to an automaton or a set of automata, it is actually meant that the operation is applied to the regular languages that are represented by the automata.

Automata are not only manipulated by the above-mentioned operations. In some cases we use the `trim` operation from the JSA library, and we have implemented our own substring operations. These operations build a new automaton from the given input automata without using any of the ‘classical’ automaton operations, as described below.

Refer to Figure 2.3, which shows an automaton that might typically arise during symbolic execution (ignore the red notations for now). Figure 2.3 has been rewritten into an equivalent set of union operations given in Figure 2.4(a).

During symbolic execution of strings, there may arise a `substring` constraint, e.g., s_1.`substring`(2, 4). We require an automata operation to apply to an automaton (such as the one in Figure 2.3) that produces all the sub-

```
cat   ∪
do    ∪
done  ∪
a*
```

(a) Original Automaton (α)

```
t   ∪
ne  ∪
a*
```

(b) Resultant Automaton (β)

Figure 2.4: Substring operations: $\alpha.\texttt{substring}(2,\, 4) = \beta$

strings starting from some concrete index, i, and ending at a different concrete index, j, where $i \leq j$. Applying this operation with $i = 2$ and $j = 4$ to Figure 2.3 produces the automaton described in Figure 2.4(b)

Our algorithm to extract a substring automaton from an input automaton can be summarised in seven steps:

1. Minimize and remove all unreachable states from the input automaton.

2. Determine all the states reachable in exactly i transitions from the start state, and call this set S.

3. Determine all the states reachable in exactly j transitions from the start state, and call this set F.

4. Discard any states that can be reached by a minimum of $j+1$ transitions.

5. Make a new start state that has an ϵ transition to every state in S.

6. Make a new accepting state that has an ϵ transition from every state in F to it.

7. Intersect the resultant automaton with an automaton representing all words of length $j - i$.

If we were to apply this to our Figure 2.3 example with $i = 2$ and $j = 4$, we would obtain the set S as $\{2, 5, 8\}$ and F as $\{7, 8\}$. There are no states that can be reached from a minimum of 5 or more transitions, so there is nothing to discard. Now we make a new start state that has an epsilon transition from it to 2, 5 and 8. Finally, an epsilon transition is added from 7 and 8 to a new

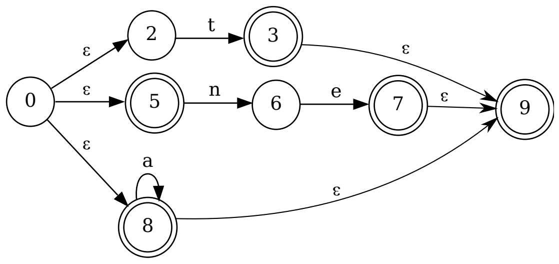


Figure 2.5: The example automata after applying the substring operation where $i = 2$ and $j = 4$.

accepting state. The resulting automaton in Figure 2.5 expresses our desired answer.

In our discussion we have omitted the infinite non-accepting state: a state for which any undefined transition would lead to. Its inclusion in our example would still lead to the same result.

Multi-track DFAs [34] are not considered in this work. In short, a Multi-track DFA is able to simulate a relation between two regular languages.

2.4 Bitvectors and SMT solvers

The Satisfiability Modulo Theories (SMT) problem is a decision problem which is expressed in first-order logic. An SMT instance is a generalised form of a boolean satisfiable (Boolean SAT) instance where sets of variables represents predicates from underlying theories. Expressing constraints in SMT tend to be more natural than Boolean SAT.

A SMT solver capable of solving these kinds of problems are able to reason about lists, arrays and bitvectors. Generally, a SMT solver is a layer on top of several third-party constraints solvers (SAT solver, integer constraint solver, etc.) and attempts to solve the given constraint by invoking the correct solver as few times as possible. One of the methods which can be used to express constraints is with bitvectors, which are defined as:

Definition 2 *A bitvector is an ordered set of bits, where each bit is either* `true` *or* `false`*. The cardinality of this set is fixed. Subsets can be obtained by* *using the form $a[i : j]$, where a is a bit-vector and i and j are both nonnegative integers where $i \geq j$. The subset $a[i : j]$ is simply the set of bits from the j-th element up to, and including, the i-th element.*

All SMT-solvers, that comply to the SMT-LIB 2 standard [1] are capable of solving constraints that contain bitvectors. If a string constraint solver has the capability of expressing its constraints in terms of bitvectors it can use one of several powerful and fast SMT solvers. With this capability the string constraint needs only to be concerned about the translation of constraints which is trivial compared to the solving of it.

A SMT solver accepts the conjunction and disjunction of bitvectors. As the definition states, the length of bitvectors need to be fixed to a constant integer. This may seem like an unnecessarily harsh restriction, given that Java string variables may grow arbitrarily long, but without such limits, decision problems may become undecidable.

Each symbolic string is represented by a bitvector with eight bits for each character in the string. The character at index i of string a, is $a[(i+1)*8-1 : (i+1)*8-8]$. The first character of the string is stored at the lowest index of the bitvector, and the last character at highest possible index.

A constraint is expressed as a conjunction (or disjunction) of constraints on the bitvector's characters. For example, given string a and its bitvector representation a_{bv} with length 32, the constraint that the first two characters of a are `ab`, is expressed as:

$$\begin{array}{l} a_{bv}[7:0] = \texttt{01100001} \wedge \\ a_{bv}[15:8] = \texttt{01100010} \end{array}$$

If the constraint is extended by including the constraint that a must end with `a` or `b`, the list of constraints becomes:

$$\begin{array}{l} a_{bv}[7:0] = \texttt{01100001} \wedge \\ a_{bv}[15:8] = \texttt{01100010} \wedge \\ (a_{bv}[31:24] = \texttt{01100001} \vee \\ a_{bv}[31:24] = \texttt{01100010}) \end{array}$$

Op	**HW**	**Hampi**	**Pex**	**Kaluza**	**JSA**	**Fujitsu**	**JPF**
`charAt`			√		△		√
`concat`	√	△	√	√	△	√	√
`contains`	√	√	√	√	△	√	√
`endsWith`	√	√	√	√	△	√	√
`equals`		△	√	√	△	√	√
`indexOf`			△		△	△	√
`length`			√	√	△	△	√
`replace`	√	√	△	√	△		△
`split`				√	△		
`startsWith`	√	√	√	√	△	√	√
`substring`			√	△	△	△	√
`trim`					△		√
`valueOf`					△	△	
`Reg.Exp.`	√	√		√	△		

Figure 2.6: Comparison between our work (JPF) and other published work. A check is full support, triangle is partial support.

Once the constraints are constructed, they are passed on to a SMT-Solver, which, if the problem is satisfiable, will return a map which represents one solution for any given variable.

SMT solvers can operate incrementally. Instead of passing the entire problem after construction, one can pass each constraint as it is built and get immediate updates on whether the problem is satisfiable or not.

For our work we used the Z3 SMT solver [5] developed by Microsoft Research. We have also considered using CVC [2] but found its bitvector solving slower than that of Z3. Because we use the the universal SMT-LIB 2 [1] specification in expressing our bitvectors and their constraints, we can easily exchange Z3 for another SMT solver.

In the rest of this thesis, we use a more user friendly-notation when it comes to bitvectors. Instead of addressing in terms of bits, we will be addressing in terms of bytes, e.g., $a[7:0]$ = `01100010` will become $a[0]$ = `b`.

The actual solving of bit-vectors is beyond the scope of this work.

2.5 Related Work

Figure 2.6 is a summary view of the comparison between each published work. This table has been compiled from what could be derived of published work. Each project in this view may have already improved since its publication.

The columns in table are as follows:

- **HW**: Short for Hooimejiers, Weimer as published in [15]
- **Hampi**: The Hampi tool developed by Kieżun et. al. [18]
- **Pex**: Developed by Tillmann et. al. at Microsoft Research [31]
- **Kaluza**: Part of the Kudzu project developed by Saxena et. al. [26]
- **JSA**: Developed by Christensen et. al. [4]
- **Fujitsu**: A symbolic execution engine also based on JPF, developed by Shannon et. al. [28]
- **JPF**: This work

A triangle indicates partial support, and check indicates full support. The reasons for the placement of the symbols on the figure, will be discussed in the following paragraphs.

A red column heading indicates an automata approach and blue column heading indicates a bitvector approach.

2.5.1 Hooimejier's Lazy approach

Hooimeijer's approach [15] consists of constructing a graph representing the constraints and variables involved, then walking through the graph using a search heuristic, and guessing solutions along the way. This does not keep track of sets of solutions, but intelligently guesses a select few possible solutions that may work, and if not will backtrack and continue.

Hooimeijer's graph is accompanied by a mapping from each string constraint to the edges and vertices involved with that constraint.

The graph exploration of Hooimeijer is an involved process. Selecting a certain few solutions with a guarantee that they are correct is tricky, also backtracking unnecessarily is difficult to avoid.

Hooimeijer does not consider operations such as `trim` and any that would be affected by symbolic integers, and we believe his theories would have to be adjusted quite dramatically in order to adapt to them.

His approach uses automata to calculate solutions, although it appears these automata are stored temporarily and need to be recomputed with backtracking.

2.5.2 HAMPI

HAMPI [18] is a specialised string constraint solver, with its own defined input grammar. It processes the input and translates the constraints to bitvector constraints which are solved by the STP SMT solver [8].

If only one symbolic string is present in the constraints and a lower and upper bound is placed on its length, it can determine the length of the symbolic string. If there are two or more symbolic strings, their lengths need to be specified by the user.

The input is translated to a simplified intermediate grammar. This is mostly to ease translation to bitvectors and to help optimise performance.

HAMPI lacks support for symbolic integers which means it does not support `charAt` and `indexOf` operations. On the other hand, it does support regular expressions and context-free grammars.

2.5.3 Kaluza

Kaluza [26] is part of a larger project (Kudzu) and is used to identify bugs in JavaScript programs. It follows the simpler approach of translating each constraint into a set of HAMPI constraints, although for concatenation it constructs a representation graph.

Due to Kaluza passing the string constraint solving mostly onto HAMPI, finding a comparison without touching on HAMPI is difficult. Due to some limitations of HAMPI Kaluza has to add some layers. For example, a graph needs to be constructed to keep track of how strings are concatenated and how their characters depend on each other.

Generally `replace` is undecidable because it may occur infinitely many times within a string. To this end, HAMPI observes a concrete execution of the software and extracts the number of times a regular expression was replaced within a string. This information is then used to force the replace operation to occur exactly the same number of times in the symbolic execution.

Kaluza, as a whole, does not support symbolic integer-string operations such as `charAt`, `indexOf` and `substring`. It does, however support regular expressions, `replace` and `split`.

2.5.4 JSA

JSA [4] uses static analysis to build a flow graph of a Java program. Then a "special" context-free grammar is defined from the model and the Mohri-Nederhof algorithm [21] is applied to obtain an approximate regular expression which expresses the set of inputs which satisfies the majority of the Java program's string constraints.

With the resulting regular expression that JSA provides, one can verify if it contains the subset of any known security vulnerabilities, such as SQL injections.

Importantly, it seems as if this approach can only handle a single symbolic string variable and cannot deal with symbolic integer inputs. Of course, this restriction severely limits the usefulness of this technique.

2.6 Overview

Symbolic execution is a technique which is able to reason about the input domain of a program. This technique can help cover a wide range of a program's state. To implement this technique we have extended JPF-symbc, which is an extension to JPF-core. Providing us with the basic blocks are automata and bitvectors. The automata approach translates and solves string operations, compared to the bitvector approach which only deals with the translating of string operations to equivalent bitvector constraints. Other published work have been divided between using automata or bitvectors when solving string constraints.

In the following section we describe our approach. How we construct the necessary path conditions from Java programs, and solve the path condition's constraints.

Chapter 3

Approach

In this section we answer the following questions:

1. How do we apply symbolic string execution to a given Java program? (Section 3.1)
2. Given a path condition that contains string and integer constraints, how do we solve it? (Section 3.2)
3. How is it possible to decide between automata solving or bitvector solving late in the process? (Section 3.3, Section 3.4)
4. Given the two approaches widely used, automata and bitvectors, how do their solving compare for a given path condition? (Section 3.5, Section 3.6)
5. How are integer constraint solving and string constraint solving integrated to work in one solver? (Section 3.7)
6. How does this work compare to other published work? (Section 3.9)

3.1 Constructing path conditions

Before any solving can start, the input needs to be constructed. Because this is a symbolic execution approach, the input is a path condition. To build this path condition, JPF-symbc executes the input Java source code.

When a Java program is executed, it is run within the Java Virtual Machine (JVM), which is a stack-based machine. JPF-core is a replacement for the standard JVM which has its own virtual machine and its own implementation of basic stack-based operations.

JPFs VM instructions are all defined within the JPF-core *Instruction factory*. JPF-symbc, which enables symbolic integer execution, extends, and overwrites, certain operations within this Instruction factory to enable the tracking of operations on symbolic variables.

A virtual machine receives a stream of bytecodes with each bytecode capable of potentially altering the program's memory. The program's memory is a stack data structure consisting of integers. Each of these integers represents either a data value or an address, and is accompanied by a set of attributes. In normal execution, these attributes are used for meta-data concerning things such as caller name, callee name, thread scheduling, etc. This attribute space is used to store the symbolic variable and/or expression that the integer value could be representing.

When a method is invoked, it pops a certain number of parameters from the top of the machine stack, and pushes at most one value on top of the stack (its return value). This modification of the stack data structure is known as the method's *execution signature*.

For this approach, JPF-symbc and JPF-core's Instruction factory is extended to 'catch' any string operations that occur during runtime, and to execute its own implementation of the string operation instead. This new implementation is responsible for two things. First, it alters the stack attributes in such a way that the symbolic variables and symbolic expressions are created and manipulated in the correct way. Secondly, it alters the actual stack values as if the original intended string operation did occur. In other words, it maintains the original's execution signature.

As an example, consider the bytecode stream in Figure 3.1. When the `isub` instruction at position 7 is executed, the stack is changed by popping the two top values, subtracting one from the other, and pushing back the result (Figure 3.2a). When symbolic execution is enabled, the top two values labelled some symbolic names (such as x and y), and the appropriate symbolic

```
int z;
if (x <= y) {
        z = y - x;
} else {
        z = x - y;
}
return z;
```

(a) Original Program

```
0: iload_0
1: iload_1
2: if_icmple 12
5: iload_0
6: iload_1
7: isub
8: istore_2
9: goto 16
12: iload_1
13: iload_0
14: isub
15: istore_2
16: iload_2
17: ireturn
```

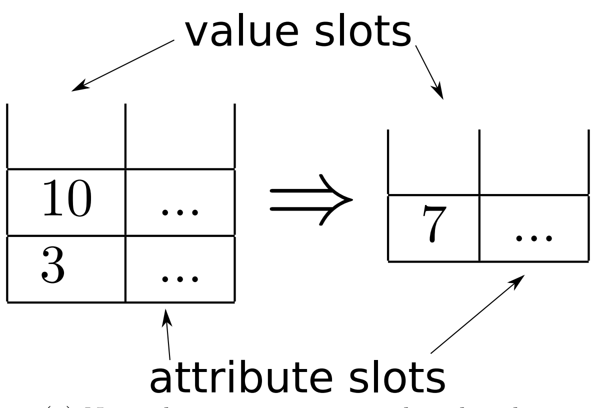
(b) Byte code

Figure 3.1: An example of a stream of bytecode

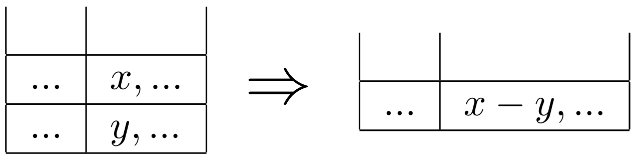
expression is pushed back $(y - x)$. Note that during symbolic execution the actual integer values of symbolic variables are ignored; this works as long as the stack frame maintains the same execution signature it would have had during normal execution. The symbolic variable names that are now stored in the attribute space of the garbage integer values can now be used by the symbolic instruction that may follow (Figure 3.2b).

Although the example is concerned with integers, building string path conditions work in the same way. For example, the operation `a.equals(b)` would place `a` and `b`'s addresses on the stack, and after the operation is executed (with normal execution), they would be replaced by a boolean value. Under symbolic execution, `a` and `b` would be popped off the stack and get the symbolic labels x and y, and some boolean value would be pushed on top along with x`.equals(`y`)` in its attributes area.

Symbolic expressions are represented as abstract syntax trees. Generally, each symbolic operation creates a new vertex to represent that operation and connects the given parameters' vertices to the created vertex.



(a) Normal execution, using only value slots



(b) Symbolic execution, using attribute slots and ignoring value slots

Figure 3.2: Stack alterations

3.2 Our approach

Before describing the details of our approach, it is worth considering a naïve solution to the problem to appreciate the obstacles and intricacies involved in the process.

Firstly, consider constraints that involve only symbolic string variables. (The problem of symbolic integers and how symbolic string constraints are dependant upon them, is considered later.)

If the solver is based on automata operations the problem seems simple. The most common string operations such as `equals`, `startsWith`, `endsWith` and `contains` all have equivalent automaton operations. Unfortunately this is not true for the negated versions of the operations.

Positive string operations

The mapping of positive string operations to automaton operations is simple. Given the string operations `equals`, `startsWith`, `endsWith` and `contains` and two symbolic string variables s_1 and s_2, the following recipes can be defined for the given string operations with the symbolic string variable parameters s_1 and s_2 (where a_i is the automaton that represents the set of s_i's solutions):

(s_1).equals(s_2)	$a_{new} := a_1 \cap a_2, s_1.\texttt{state} := a_{new}, s_2.\texttt{state} := a_{new}$.
(s_1).startsWith(s_2)	$a_{new} := a_1 \cap (a_2 \oplus .*), s_1.\texttt{state} := a_{new}$,
	$a_{new} := a_2 \cap (\texttt{startsWith}(a_1)), s_2.\texttt{state} := a_{new}$.
(s_1).endsWith(s_2)	$a_{new} := a_1 \cap (.* \oplus a_2), s_1.\texttt{state} := a_{new}$,
	$a_{new} := a_2 \cap (\texttt{endsWith}(a_1)), s_2.\texttt{state} := a_{new}$.
(s_1).contains(s_2)	$a_{new} := a_1 \cap (.* \oplus a_2 \oplus .*), s_1.\texttt{state} := a_{new}$,
	$a_{new} := a_2 \cap (\texttt{allSubstrings}(a_1)), s_2.\texttt{state} := a_{new}$.

Strictly speaking the intersection and concatenation operation are only defined for regular languages and not for automata. When we say that automata are intersected or concatenated, the operations are in actual fact being applied to the regular languages that are represented by the respective automata.

To put the given recipes in English:

- For the `equals` operation: intersect the two automata representing the two symbolic string variables (s_1 and s_2) and produce a new temporary automaton a_{new}. Assign the solution sets of s_1 and s_2 to a_{new}.

- For the `startswith` operation: intersect the automaton a_1 (representing the symbolic string variable s_1) with an automaton that accepts all words that start with some word accepted by a_2. Assign this intersection to s_1's set of solutions. For the second step, intersect the automaton of the second symbolic string variable with the automaton of the first symbolic string variable in such a way that it consists of all possible prefixes from a_1. Assign this product to be s_2's set of solutions.

- For the `endswith` operation: intersect the automaton a_1 (representing the symbolic string variable s_1) with an automaton that accepts all words that end with some word accepted by a_2. Assign this intersection to s_1's set of solutions. For the second step, intersect the automaton of the second symbolic string variable with the automaton of the first symbolic string variable in such a way that it consists of all possible suffixes from a_1. Assign this product to be s_2's set of solutions.

```
1  a.startsWith('hello')
2  a.equals(b)
3  a.contains('a')
```

Figure 3.3: A simple set of string operations

- For the `contains` string operation: Intersect the automaton (a_1) representing the first symbolic string variable (s_1) in such a way that it only contains the words that contain the words from the automaton (a_2) representing the second symbolic string (s_2). Assign this product to s_1's set of solutions. For the second step, intersect the automaton of the second symbolic string variable with the automaton of the first symbolic string variable in such a way that it only contains all possible substrings from a_1. Assign this product to be s_2's set of solutions.

It may be necessary to iterate through the recipes several times until all the involved automata converge.

As an example, consider Figure 3.3. There are two symbolic strings a and b, and two constant strings `hello` and `a`. Let A_a represent a's automaton and A_b represent b's automaton:

1. Initiate both automaton to the universal automaton (i.e., the automaton accepts all possible words).

2. **Line 1** Intersect the automaton A_a (currently equivalent to `.*`) with `hello.*` giving `hello.*`.

3. **Line 2** Intersect the automata A_a and A_b (`hello.*` and `.*` respectively), producing `hello.*`, assign this to both automata.

4. **Line 3** Intersect the automata of A_a (`hello.*`) with `.*a.*`, producing `hello.*a.*`.

With the results from these steps A_a is `hello.*a.*` and A_b is `hello.*`. However, this does not satisfy line 2 of Figure 3.3. If the steps are repeated with A_a and A_b's initial value `hello.*a.*` and `hello.*` it would lead to the values of A_a and A_b being changed. However, since only intersection is used

1 a.**startsWith**(‘`hello`’)
2 a.**contains**(‘`a`’)
3 b.**contains**(‘`a`’)
4 ¬ a.**equals**(b)

Figure 3.4: A variation of Figure 3.3 which causes the simplistic approach to break down

there will be some amount of repetition where each automaton converge, and the iteration can be stopped.

Allowing the automata to converge for the given example will lead to A_a and A_b both being equal to `hello.*a.*`. If any word accepted by these automata are assigned to both a and b our string operations would lead to a *true* evaluation.

Negative string operations

If we negate each line of Figure 3.3, the first and third line of Figure 3.3 would still be simple to solve (simply take the complement of the automata representing `hello` and `a`), but the second line presents a problem.

To illustrate the difficulty more clearly, consider the operations in Figure 3.4:

Observe that there is at least one solution, namely a as `helloa` and b as `a`. After translating the string operations 1, 2 and 3 A_a will be `hello.*a.*` and A_b will be `.*a.*`. At this point, it is very important to note that A_a is a subset of A_b. Continuing to line 4, if we were to invert A_b (to obtain the automaton that accept all words not accepted by A_b) and intersect it with A_a, it would give an empty automaton. This will lead the algorithm to believe that there is no solution for the string operations.

To understand why the naive approach did not work on this occasion, consider the visual representation of the automata in Figure 3.5, where A_a is area 1, A_b is area 2 and the universal automaton is area 3. If the inverse of A_b is taken, in other words the entire area of $3 - 2$ and intersected with area 1 it would give no result because the two areas do not overlap.

To overcome this problem our more sophisticated solution postpones the

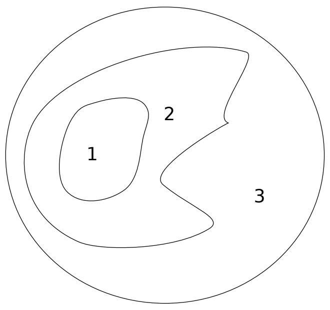


Figure 3.5: Diagram representing automata

`notEquals` operation until the other operations have converged to a solution. It then carefully selects a subset of the solution that satisfies the `notEquals` constraints. This is described in detail in Section 3.6.1.

Success is achieved easily if a naive approach to the bitvectors approach is considered. Once again, we define a recipe to translate each string operation into a set of bitvector operations. As before, the string operations considered are: **equals**, **startsWith**, **endsWith** and **contains**. The negation of these string operations are as easy as negating bitvector constraints. Because only translation of the string operation needs to be achieved and no solving of it, the problem is simplified (let $b_{i,j}$ represent the j-th character of string variable s_i; b_i refers to all characters of string variable s_i):

$$\begin{array}{ll}
(s_1)\textbf{.equals}(s_2) & b_1 = b_2. \\
(s_1)\textbf{.startsWith}(s_2) & b_{1,1} = b_{2,1}, b_{1,2} = b_{2,2}, \ldots \\
(s_1)\textbf{.endsWith}(s_2) & b_{1,l-1} = b_{2,l-1}, b_{1,l-2} = b_{2,l-2}, \ldots \\
(s_1)\textbf{.contains}(s_2) & b_{1,i} = b_{2,j}, b_{1,i+1} = b_{2,j+1}, \ldots \forall i, j
\end{array}$$

While automata can represent strings of arbitrary length, the length of the bitvectors in these recipes must be known before the constraints can be solved: This leads naturally to the case of mixed string and integer constraints.

Combining symbolic strings and integers

Both automaton and bitvector approaches suffer to some extent from a lack of symbolic integer understanding. Both are limited in some way when encountering symbolic integers and, because we want to compare the two approaches, we need to find common ground.

```
a.length() > b.length()
a.concat(b).charAt(3) == 'd'
```

Figure 3.6: A simple program with string operations

A bitvector's length needs to be specified as a constant integer during translation. If the length is symbolic, a direct translation is impossible. As mentioned above automata however, are able to handle a dynamic length symbolic string. However, both fail when intricate symbolic integer operations and dependencies are involved. For example, consider the program in Figure 3.6 First, the length of symbolic string a must be larger than the length of the symbolic string b. Second, the concatenation of a and b must contain the character '`d`' at the 3rd index.

Although Figure 3.6 has very few symbolic integers (two, one for each symbolic string length), it does imply intricate dependencies. If the length of a is larger than 3, then the `charAt` constraint applies to a if however a's length is smaller or equal to 3, then it applies to b. This shows that the integer solutions for lengths imply which constraints apply to which symbolic string variables.

The dependencies between string and integer constraints, cannot be ignored as some previous work has done [15, 4]. Substituting a proper integer solver with custom made limited solver is also not desirable [28]. We propose to replace all symbolic integers with concrete integers as determined by a best guess of an integer solver, and then to solve the string constraints for the fixed integer values. If the values lead to unsatisfiability, more integer constraints are generated and the next best guess is used. This continues until satisfiability is reached, or no more integer guesses are possible.

3.3 General strategy

For our approach we followed a general strategy:

1. Translate a given path condition into an intermediate form

2. Solve, or at least simplify, the intermediate form

3. Replace any symbolic integer with a *best guess* concrete integer

4. Translate the intermediate form to automata operations or bitvector constraints.

5. If the translation fails, repeat with a different set of best guess concrete integers

6. Terminate if the translation finds a satisfiable assignment, until no more best guess integers are available, or until a specified time limit expires.

The strategy is more formally presented as an algorithm in Figure 3.7. Line 2 creates an intermediate form of the path condition. This intermediate form is called the *string graph*. Line 4 indicates that the algorithm will loop until a satisfiable solution is found, or until a time-out has been reached. Within that loop, line 5 provides best guess concrete integers and line 6 translates the string graph (propagated with the concrete integers) into either automata or bitvectors.

The *best guess* concrete integers are simply the integer solutions that satisfy all integer constraints and possibly all integer-string constraints. Only when the string constraints are considered during the translation phase (Step 4 above) is it possible to determine if the current *best guess* is satisfying all integer-string constraints.

The initial part of the algorithm is concerned with building a string graph representation and preparing the integer constraint solver to provide the first best guess. This is followed by a two-step iterative process: (1) the integer JPF constraint solver is invoked to take care of integer, real, and boolean constraints, and (2) if this is successful, some of the values obtained (those involved in the mixed constraints) are propagated to the string graph and a string constraint solver is invoked to solve them. In our case, STRINGSOLVER is either AUTOMATONSOLVER or BITVECTORSOLVER (although it is possible to plug in any other string solvers with little work). It may be that the string graph constraints are not satisfiable for the given values, in which case the string solver adds new constraints to the path condition and the process is repeated.

```
SOLVE($PathCondition\ pc$)
 1  $string\ graph\ sg$
 2  $(pc, sg) \leftarrow$ BUILDSTRING GRAPH($pc$)
 3  boolean $sat \leftarrow$ false
 4  while $\neg sat \wedge \neg$timeout:
 5      $(sat, pc, sg) \leftarrow$ INTEGERSOLVER($pc, sg$)
 6      if $sat$: $(sat, pc, sg) \leftarrow$ STRINGSOLVER($pc, sg$)
 7      if $pc$ unchanged: break
 8  return $sat$

BUILDSTRING GRAPH($PathCondition\ pc$)
 9  $string\ graph\ sg \leftarrow \emptyset$
10  for string or mixed constraint $c \in pc$:
11      $sg \leftarrow sg \cup$ HYPEREDGE($c$)
12  return PREPROCESS($pc, sg$)
```

Figure 3.7: Core algorithm

Even if the string problem is decidable, it is possible that the combination of the integer solver and the string solver are not able to reach a conclusion on its satisfiability. To guarantee that the process terminates, an upper limit is placed on the length of the symbolic strings; any satisfiable path condition that requires a string of longer length then the defined limit will not be solved and the algorithm concludes that the constraints are not satisfiable. A time limit can also be specified to prevent the solver from spending too much time on unnecessarily intricate path conditions. If the time limit is exceeded the path condition is classified as not satisfiable.

The subsections that follow describe the string graph, its construction and preprocessing. Thereafter, a description of the approach used to translate a given string graph to automata operation or bitvectors constraints is given. And finally, how new constraints are generated when the string constraints are not satisfiable, and how this information is fed back to the integer constraint solver.

3.4 String graph

In general, constraints can be classified as either integer, string, mixed integer-string, or other (such as real or boolean). In our implementation, string and mixed constraints are represented in a special kind of graph:

Definition: A *string graph* is a labelled directed hypergraph $H = (X, E, \Omega)$ where the set of vertices $X = I \cup S$ is the union of two disjoint sets I and S that represent integer and string variables, respectively. The set Ω denotes string operations, and the set of directed labelled hyperedges is $E \subseteq E_1 \cup E_2 \cup E_3 \cup E_4$ where $E_n = \Omega \times X^n$. In other words, each labelled hyperedge is a tuple where the first component is an operation, and the other components are vertices.

The vertices in I and S correspond to either constant values, or integer or string variables. As explained below, extra vertices of either kind are sometimes added to the string graph. Hypergraphs are not essential to the operation of the algorithm in the sense that it does not rely on any particular property unique to hypergraphs. They are merely a convenient data structure; other graph-based approaches are discussed in Section 3.9.

If a string operation returns a new string variable, that variable is called a *destination variable*. The vertex representing it is called a *destination vertex*. Any other string variable is called a *source variable* and its vertex is called a *source vertex*

3.4.1 Construction

Each string or mixed constraint of the path condition contributes exactly one hyperedge to the string graph, and it is constructed constraint by constraint. Predicate operations (those that return `boolean`) are mapped straightforwardly to hyperedges. For example, the constraint s_1.`equals`(s_2) contributes the hyperedge (`equals`, s_1, s_2). For other, transformational operations (those that return `char`, `integer`, or `String`) a new auxiliary variable is introduced to represent the result. This variable is added as a vertex to the string graph, which allows the hyperedge to be added as in the case of predicate operations.

A more substantial example is shown in Figure 3.8. Circles depict vertices,

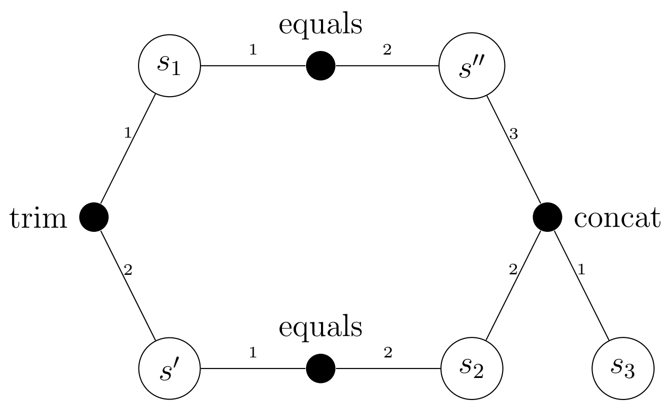


Figure 3.8: Example of a string graph

dots and lines depict hyperedges. Each hyperedge is labelled with its operation, and small numbers indicate the order of its component vertices. The string graph shown here corresponds to the constraints

$$s_1.\texttt{trim}().\texttt{equals}(s_2) \wedge s_1.\texttt{equals}(s_3.\texttt{concat}(s_2)).$$

The string variables s_1, s_2, and s_3 appear as vertices of S. For this graph, $\Omega = \{\mathsf{trim}, \mathsf{equals}, \mathsf{concat}\}$. The `trim` operation leads to the introduction of auxiliary variable s' which is added to S. The operation itself is a hyperedge (trim, s_1, s') labelled with the trim element of Ω and connected to vertices s_1 and s'. Similarly, the `concat` operation in the second constraint leads to the introduction of auxiliary variable s'' and the hyperedge $(\mathsf{concat}, s_3, s_2, s'')$. Finally, the two `equals` operations are responsible for two more hyperedges.

Following is a list of possible elements in Ω and how they operate.

- **CharAt**

  $(\mathsf{charAt}, s_1, i_1, i_2)$ where s_1 is some existing symbolic string, i_1 is some existing symbolic integer depicting the index of the character and i_2 is some existing symbolic integer which depicts the value of the character.

  charAt is the operation where a character i_2 is specified to appear at index i_1 in string s_1.

- **Concat**

  $(\mathsf{concat}, s_1, s_2, s')$ where s_1 and s_2 are existing symbolic strings, and s' is a new symbolic string depicting the result.

  concat specifies a string s''s first part to the consist of s_1 and the rest of s_2.

- **Contains**

  $(\mathsf{contains}, s_1, s_2)$ where s_1 and s_2 are existing symbolic strings.

  contains is the operation which specifies s_2 to be contained within s_1

- **EndsWith**

  $(\mathsf{endsWith}, s_1, s_2)$ where s_1 and s_2 are existing symbolic strings.

  endsWith is the operation which specifies s_1 to end with s_2

- **Equal**

  $(\mathsf{equal}, s_1, s_2)$ where s_1 and s_2 are existing symbolic strings.

  equal is the operation which specifies s_1 and s_2 to have the same value.

- **IndexOf**

  1. $(\mathsf{indexOf}, s_1, s_2, i')$ where s_1 and s_2 are existing symbolic strings and i' is the resulting symbolic integer.
  2. $(\mathsf{indexOf}, s_1, s_2, i_1, i')$ where s_1 and s_2 are existing symbolic strings and i_1 is an existing symbolic integer. i' is the resulting symbolic integer.
  3. $(\mathsf{indexOf}, s_1, i_1, i')$ where s_1 is some existing symbolic string and i_1 is some existing symbolic integer. i' is the resulting symbolic integer.
  4. $(\mathsf{indexOf}, s_1, i_1, i_2, i')$ where s_1 is some existing symbolic string and i_1 and i_2 are existing symbolic integers. i' is the resulting symbolic integer.

If $i' \geq 0$

In cases 1 and 2, s_2 is being specified to occur at index i' in s_1, where no occurrence of s_2 may appear between index 0 and index $i' - 1$ in s_1. Case 2 replaces the index 0 with index i_1.

Cases 3 and 4 specifies the character depicted by i_1 to be at index i' in s_1, where no occurrence of the character may appear between index 0 and index $i' - 1$ in s_1. Case 4 replaces the index 0 with index i_2.

If $i' < 0$

In cases 1 and 2, s_2 must not be present within s_1. In cases 3 and 4 the character depicted by i_1 must not be present within s_1.

- **LastIndexOf**

  1. $(\mathsf{lastIndexOf}, s_1, s_2, i')$ where s_1 and s_2 are existing symbolic strings. i' is the resulting symbolic integer.
  2. $(\mathsf{lastIndexOf}, s_1, s_2, i_1, i')$ where s_1 and s_2 are two existing symbolic strings and i_1 is some existing symbolic integer. i' is the resulting symbolic integer.
  3. $(\mathsf{lastIndexOf}, s_1, i_1, i')$ where s_1 are existing symbolic string and i_1 are existing symbolic integer. i' is the resulting symbolic integer.
  4. $(\mathsf{lastIndexOf}, s_1, i_1, i_2, i')$ where s_1 are existing symbolic string and i_1 and i_2 are existing symbolic integers. i' is the resulting symbolic integer.

If $i' \geq 0$

In cases 1 and 2, s_2 is being specified to occur at index i' in s_1, where no occurrence of s_2 may appear between the end of s_1 and index i'. Case 2 replaces the end of s_1 with index i_1.

Cases 3 and 4 specifies the character depicted by i_1 to be at index i' in s_1, where no occurrence of the character may appear at the end of s_1 and index i'. Case 4 replaces the end of s_1 with index i_2.

If $i' < 0$

In cases 1 and 2, s_2 must not be present within s_1. In cases 3 and 4 the character depicted by i_1 must not be present within s_1.

- **notContains**

  $(\mathsf{notContains}, s_1, s_2)$ where s_1 and s_2 are existing symbolic strings.

  notContains specifies s_2 to be in not a substring of s_1.

- **NotEndsWith**

  $(\mathsf{notEndsWith}, s_1, s_2)$ where s_1 and s_2 are existing symbolic strings.

  notEndsWith specifies s_2 not to end with s_1.

- **notEqual**

  $(\mathsf{equal}, s_1, s_2)$ where s_1 and s_2 are existing symbolic strings.

  equal specifies s_1 and s_2 to have different values.

- **NotStartsWith**

  $(\mathsf{notStartsWith}, s_1, s_2)$ where s_1 and s_2 are existing symbolic strings.

  notStartsWith specifies s_2 not to be at the start of s_1.

- **StartsWith**

  $(\mathsf{startsWith}, s_1, s_2)$ where s_1 and s_2 are existing symbolic strings.

  startsWith specifies s_1 to begin with s_2.

- **Substring**

  1. $(\mathsf{substring}, s_1, i_1, s')$, where s_1 are existing symbolic string, and i_1 is an existing symbolic integer. s' is the result of the constraint.
  2. $(\mathsf{substring}, s_1, i_1, i_2, s')$, where s_1 is some existing symbolic string, and i_1 and i_2 are existing symbolic integers. s' represents the result of the constraint.

substring, when applied, extracts a substring of characters between i_1 and the end of s_1 (or up to the index i_2, if case 2). This substring is represented by s'.

- **Trim**

  (trim, s_1, s'), where s_1 is some existing symbolic string and s' represents the result of the trim operation.

  trim removes trailing and leading spaces from s_1 to produce s'.

3.4.2 Preprocessing

To avoid unnecessary overhead we simplify the string graph before involving the string solver. Certain classes of subgraphs within the string graph can be simplified, or, by their presence, prove that the string graph is trivially unsatisfiable.

Therefore, before translating the string graph into an input that the string solver can understand, we apply a series of algorithms that identify a defined set of subgraphs in order to prove unsatisfiability or to simplify them.

In our current implementation these are simple small subgraphs consisting of two edges and a common source vertex, or consisting of the single hyperedge $(\mathsf{equal}, s_1, s_2)$ and the two vertices it connects. This latter subgraph can be simplified by removing s_2, and merging the incoming and outgoing edges of s_2 and s_1.

After applying the series of subgraph modifications to the string graph, the preprocessor can determine if the string graph is definitely unsatisfiable (without needing to consult the integer or string solver) or possibly satisfiable. With this we hope to minimise the number of times needed to consult the third party integer and string solver. We are invoking the integer and string solver only in instances were the set of constraints are "difficult" to solve.

During the latter phase of preprocessing we also prepare the integer solver to produce its first solutions of the symbolic integers (to propagate the string graph with). These solutions are called *best guesses*, because they are not necessarily the correct integer solutions that lead to satisfiability Once again we need to identify certain subgraphs within the string graph which signifies certain integer constraints. For example, the subgraph consisting of the hyperedges $(\mathsf{startsWith}, s_1, \text{‘}abc\text{’})$ and $(\mathsf{charAt}, s_1, 0, i_1)$ would indicate that the integer constraint of $i_1 = \text{‘}a\text{’}$ has to be satisfied, and so we can pass this implicit constraint explicitly to the integer solver.

Simplifying

The following steps are applied to simplify the string graph:

1. For every $(\mathsf{equal}, s_1, s_2)$ we remove the hyperedge and the vertex denoting s_2, and reconnect any edges involving s_2 to use s_1 in its place.
2. For every hyperedge where all elements except one are constants, replace the one symbolic element with the only possible solution (if only one solution is possible). e.g., $(\mathsf{concat}, \text{‘}ab\text{’}, \text{‘}c\text{’}, s_3)$ is simplified to the hyperedge $(\mathsf{concat}, \text{‘}ab\text{’}, \text{’}c\text{’}, \text{‘}abc\text{’})$.

We step through each edge of the graph, modifying it according to the above steps. The graph may be changed during such an iteration due to the fact that step 2 may introduce more constants, and thus other edges (previously considered, but did not satisfy step 2) may now satisfy step 2. Therefore, these iterated are applied until no more modifications are made to the graph.

Detecting unsatisfiability

The following subgraphs indicate definite unsatisfiability:

1. Any pair of opposing boolean string constraints (for example equals and $\mathsf{notEquals}$) between the same two vertices.
2. Any hyperedge where all the elements are constants but do not satisfy the hyperedge.
3. A pair of constraints of the form $\{(e, s_a, s_b), (e, s_a, s_c)\}$ where e is from the set of $\{\mathsf{endsWith}, \mathsf{startsWith}\}$, s_b and s_c are not equal but constant.

4. If the integer solver is unable to generate at least one best guess.

The possible pair of opposing boolean string constraints are:

$$\{\mathsf{equals}, \mathsf{notequals}\}$$
$$\{\mathsf{startsWith}, \mathsf{notstartswith}\}$$
$$\{\mathsf{endsWith}, \mathsf{notendswith}\}$$
$$\{\mathsf{contains}, \mathsf{notcontains}\}$$

It is also possible to identify indirect opposing constraints that arise during simplification of the string graph. An example of these indirect opposing constraints is a circular set of equality between vertices, where one edge is non-equal. A concrete example is shown in Figure 3.9.

Not all unsatisfiability will be detected by this level of checking. However, it is possible to detect a certain class of unsatisfiability which may have taken the translation phase longer to find. Also, time is saved by not invoking the translation if there is no satisfiable solutions.

Deriving integer constraints

Not only does each hyperedge in the string graph imply an integer constraint, but each possible pair of hyperedges may also imply integer constraints. (As shown in Table 3.1.)

We identify such pairs and pass the integer constraints explicitly to the integer solver. The use of such pairs amount to a heuristic. It is possible to

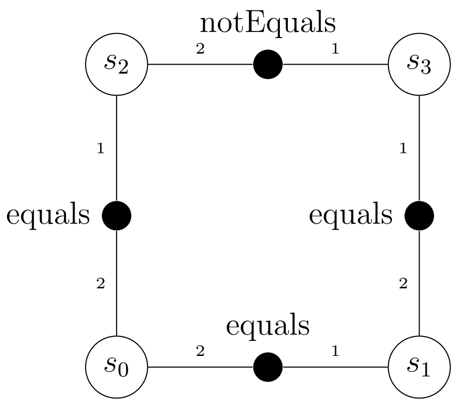


Figure 3.9: Circular set of constraints which will lead to no satisfiable solutions

Subgraph	**Constraint**
$\{(\mathsf{charAt}, s_a, c_1, i_1), (\mathsf{charAt}, s_a, c_2, i_2)\}$	$i_1 \neq i_2 \lor c_1 = c_2$
$\{(\mathsf{indexOf}, s_a, s_b, i_1), (\mathsf{charAt}, s_a, c, i_2)\}$	$i_1 \neq i_2 \lor s_b.\mathtt{charAt}(0) == c$
$\{(\mathsf{indexOf}, s_a, s_b, i_1), (\mathsf{indexOf}, s_a, s_b, i_2, x)\}$	$x \geq i_1$

Figure 3.10: Integer constraints derived from certain pairs of hyperedges

Table 3.1: String operations, hyperedges, and constraints

Java expression	Hyperedge	New constraints
For each non-constant vertex $s_i \in S$		$\ell_i > 0$
$s_a.\mathtt{charAt}(n)$	$(\mathsf{charAt}, s_a, n, c)$	$\ell_a \geq n$
$s_a.\mathtt{concat}(s_b)$	$(\mathsf{concat}, s_a, s_b, s_x)$	$\ell_x = \ell_a + \ell_b$
$s_a.\mathtt{indexOf}(s_b)$	$(\mathsf{indexOf}, s_a, s_b, x)$	$(x = -1) \lor (\ell_a \geq \ell_b + x)$
$s_a.\mathtt{lastIndexOf}(s_b)$	$(\mathsf{lastIndexOf}, s_a, s_b, x)$	$(x = -1) \lor (\ell_a \geq \ell_b + x)$
$s_a.\mathtt{substring}(n)$	$(\mathsf{substring}, s_a, n, s_x)$	$\ell_a = n + \ell_x$
$s_a.\mathtt{substring}(n, k)$	$(\mathsf{substring}, s_a, n, k, s_x)$	$(\ell_a \geq n + k) \land (\ell_x = k)$
$s_a.\mathtt{trim}$	$(\mathsf{trim}, s_a, s_x)$	$\ell_a \geq \ell_x$
$s_a.\mathtt{contains}(s_b)$	$(\mathsf{contains}, s_a, s_b)$	$\ell_a \geq \ell_b$
$s_a.\mathtt{endsWith}(s_b)$	$(\mathsf{endsWith}, s_a, s_b)$	$\ell_a \geq \ell_b$
$s_a.\mathtt{startsWith}(s_b)$	$(\mathsf{startsWith}, s_a, s_b)$	$\ell_a \geq \ell_b$

exhaustively define all possible combinations but we have limited our implementation to those that occur most frequently. Due to the sheer number of possible pairs (253 possible pairs to be exactly), we will confine ourselves to only a few examples considered in this work. Of the 253 possible pairs, we have implemented 80.

To make this even more concrete, consider Figure 3.11. The figure shows the string graph which resulted from the two constraints $\{(\mathsf{charAt}, s_0, 5, x), (\mathsf{indexOf}, s_0, s_1, 5)\}$. These constraints are depicted by the black edges within the figure. The dashed line hyperedge is an implied constraint. This implied constraint is derived from the fact that the character x and the first character of s_1 will overlap.

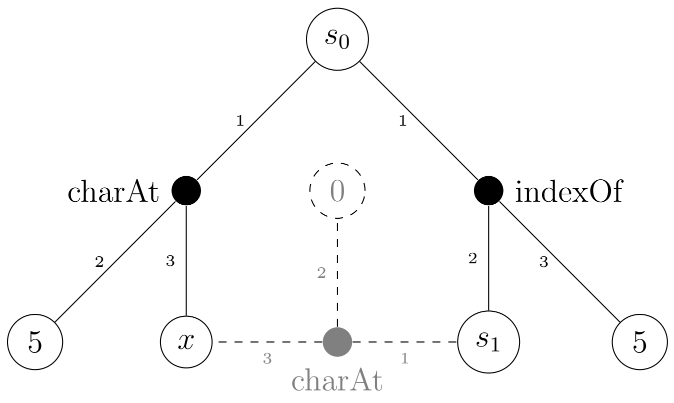


Figure 3.11: Given the constraints that s_0 must have have the character x at position 5, and s_1 at index 5; implies that s_1 must have x at its index 0.

3.5 Common ground

We decide at the last possible moment on whether automata or bitvectors will be used in the solving of the path condition. This requires generalisations during the translation phase which causes both approaches to suffer from the other's limitations. Some of these limits are:

- **Fixed lengths:** Automata are capable of handling infinite strings, but each bitvector requires a constant integer as length.
- **ASCII values:** Automata are able to store Unicode strings without any significant time penalty. Bitvectors on the other hand pay a big price for string encodings larger than 8 bits. In theory one could make all Unicode characters 32 bit wide. It is only a low-level encoding and does not affect high level string constraints.
- **Fixed constant indices:** If only the bitvector approach were used, the SMT-solver could be used to set certain of the indices as uninterpreted functions. Unfortunately automata do not have this power, so we have to solve the indices before hand.

Due to the approach enforcing these limitations early on, it is able to decide between the two approaches at a later stage. A side-effect of this is that the solver is able to switch between approaches for a given set of path conditions.

This can be used in future to invoke the solver best suited for the particular path condition. Also, this enables our framework to easily incorporate alternative approaches (e.g., transducers) in the future.

3.6 Translating

A preprocessed string graph is explored edge by edge during the translation phase. For each type of edge, there is a corresponding recipe which tells the translator how to alter the source and destination of that edge, depending on whether the automata approach or bitvectors approach are being used.

The sequence in which the edges are explored are completely arbitrary. There is no ordering, sorting or forcing on the sequence of edges. Ordering of edges are considered for future work.

The translation phase has three inputs:

- The approach to use (automata or bitvector)
- A string graph
- A set of constant integers with which each symbolic integer is replaced with (named **best guess**).

The translation phase may conclude that the current string graph (with its best guess) is not satisfiable. In this case the integer constraint solver is notified that the current best guess solutions for the integer variables did not work. If there is more possible integer solutions the translation is invoked again, until the string graph is found to have a satisfiable solution, or until there is no more possible integer solutions. This entire process is called the interchange, and is described in Section 3.7. This section lists and explains the translation recipes used, and the reasoning behind them.

This work contributes a novel approach of being able to decide between automata and bitvectors, compared to previously published string constraints solvers which were written for only one approach. One of our outcomes is to be able to compare the theoretical power and practical performance between the two approaches.

```
AUTOMATONSOLVER($PathCondition\ pc$,
        $string\ graph\ sg = (I \cup S, E, \Omega)$)
 1  for string vertex $s_i \in S$:
 2      $FiniteAutomaton\ M_i \leftarrow [\ell_i]$
 3  $W \leftarrow E$
 4  while $W$ contains a positive hyperedge:
 5      remove any positive hyperedge $e$ from $W$
 6      for $s_i$ connected to $e$:
 7          update $M_i$ based on the recipe for $e$
 8          if $M_i$ is empty:
 9              $pc \leftarrow pc \cup$ FRESHCONSTRAINTS($pc, sg$)
10              return (false, $pc, sg$)
11          else if $M_i$ has changed:
12              $W \leftarrow W \cup \{$all hyperedges connected to $s_i\}$
13  return CHECKNEGATIVEEDGES($pc, sg, (M_1, M_2, \ldots)$)
```

Figure 3.12: The automaton-based solver algorithm

3.6.1 Translation to automata

Finite automata are a natural choice for the representation of string variables: the regular language of an automaton is a set of words in the same way that a string variable can store any one of a set of words that satisfy the constraints. Constraints themselves corresponds to regular language operations (such as intersection and union), as we explain shortly.

Although we are aware of at least one automaton string constraint solver which is capable of giving us a satisfiable/unsatisfiable result [15] (referred to as the Hooimeijer approach), we chose not to use it, because we feel it is limited in its capabilities of handling symbolic integers. Instead, our implementation uses a home-made solver that takes the whole of the string graph as input, iteratively translates the hyperedges, and uses a standard automaton toolkit (JSA [4] enhanced with some of our own routines) to produce its results.

The automaton-based solution is shown in Figure 3.12. An automaton M_i is constructed for each string vertex s_i of the string graph. (For clarity, the notation $[n]$ denotes the automaton for the regular language $\cdot^n$, and $[cm]$ for the regular language c^m.) The ℓ_i string-length variables are instantiated by the integer constraint solver, and hence, for each automaton constructed in line 2, all of its words have a fixed length. Of course, different automata may have words of different lengths.

As the hyperedges are processed, the automata are modified to reflect the effect of the constraint. The exact nature of the modification depends on the constraint. For example, given hyperedge $(\mathsf{contains}, s_a, s_b)$ corresponding to the constraint $s_a.\mathtt{contains}(s_b)$, suppose that M_a is the automaton for vertex s_a, and M_b for s_b. The updated M'_a and M'_b are

$$\begin{aligned} M'_a &:= M_a \cap ([*] \oplus M_b \oplus [*]) \\ M'_b &:= M_b \cap \mathit{substrings}(M_a, 0, \infty). \end{aligned}$$

In other words, whatever set of words M_a currently accepts, it is restricted to those words that contain s_b as a substring. Correspondingly, whatever set of words M_b accepts is restricted to substrings of the words of M_a.

Similar recipes for other operations are shown in the centre column of Table 3.2. The notation $M \overset{\cap}{:=} X$ is shorthand for $M' := M \cap X$, $\oplus$ denotes language concatenation, $\mathit{substrings}(M, i, j)$ is an operation that returns all possible subwords of words of M starting at the ith character and ending before the jth, $\mathit{suffixes}(M)$ and $\mathit{prefixes}(M)$ return all possible suffixes and prefixes of words of M, respectively, and $\mathit{trim}(M)$ returns all words of M without leading and trailing whitespace.

Unfortunately, the automaton approach is unable to generate solutions for negative constraints (notEquals, notContains, notStartsWith, and notEndsWith). Suppose that M_1 and M_2 correspond to two strings that are constrained to be unequal. Three cases arise: (1) if both automata accept only a single word, the inequality is satisfiable if and only if the words differ; (2) if only one of the automata accepts a single word, the word can be removed from the language of the other automaton and the inequality is satisfiable; (3) in all other cases, the constraint is satisfiable and neither automaton needs to be modified. Unfortunately, this logic begins to break down when more than two automata are involved and other kinds of negative constraints are added. In short, it is not feasible to impose these constraints on the automaton level. The hyperedges are therefore partitioned into positive and negative hyperedges, the latter being edges of one of the four kinds mentioned at the start of this paragraph.

The main body of the algorithm in Figure 3.12 follows a typical work list pattern. Initially W contains all hyperedges. The positive hyperedges are removed one-by-one and processed. Any change to M_i causes those hyperedges connected to s_i to be placed in the work list again. Eventually, though, the algorithm must terminate because all of the automaton modifications are of the form $M' := M \cap X$. In short, the language of each automaton either stays the same or is restricted during each assignment. When a fixed point is reached, each automaton contains exactly those words that would satisfy the constraints.

If during any assignment an automaton is reduced to the empty language, the string graph and the corresponding constraints are known to be unsatisfiable. When this happens, new integer constraints are generated and control returns to the repeat–until loop of procedure SOLVE (in Figure 3.7). Since the same thing happens in the case of bitvectors, the new constraints are described in Section 3.7.

If control reaches line 13 of the AUTOMATONSOLVER routine, the M_i's contain solutions that satisfy all of the positive constraints. All that remains is to check for and find words that also satisfy the negative constraints. Because the words of each automaton have a fixed length, one solution would be to simply enumerate all the words of all the automata until a combination is found that satisfies the negative constraints. This is effectively what happens in Figure 3.13 where we build a set S_i per automaton M_i where each set contains k words. k is determined to be the number of automata involved in the negative constraints. Once we have these sets we step through each possible combination until a satisfying set of solutions are found.

Figure 3.13 gives a description of the algorithm used to find assignments that will satisfy the negative constraints. $sol(M, k)$ is a function which gives the first k words in automata M, t is a tuple which represents possible set of assignment for each vertex s_i and $verify(sg)$ is a function which checks if all constraints are satisfied with each vertex's assignment.

Table 3.2: Hyperedge recipes for the automata-based approach and hyperedge constraints for the bitvector approach

Hyperedge	Recipes	Constraints
(charAt, s_a, n, x)	$M_a \mathrel{:\overset{\cap}{=}} [n] \oplus \{x\} \oplus [*]$	$s_a[n] = x$
(concat, s_a, s_b, s_x)	$M_a \mathrel{:\overset{\cap}{=}} \mathit{substrings}(M_x, 0, \ell_a)$	$(s_a = s_x[\ell_a : 0]) \wedge (s_b = s_x[\ell_x : \ell_a])$
	$M_b \mathrel{:\overset{\cap}{=}} \mathit{substrings}(M_x, \ell_b, \infty)$	
	$M_x \mathrel{:\overset{\cap}{=}} M_a \oplus M_b$	
(indexOf, s_a, s_b, x)	$M_a \mathrel{:\overset{\cap}{=}} ([x] \cap \neg M_b) \oplus M_b \oplus [*]$	$(s_a[x + \ell_b : x] = s_b) \wedge \left(\bigvee_{0 \leq i < x} s_a[i + \ell_b : i] \neq s_b\right)$
	$M_b \mathrel{:\overset{\cap}{=}} \mathit{substrings}(M_a, c, c + \ell_b)$	
(lastIndexOf, s_a, s_b, x)	$M_a \mathrel{:\overset{\cap}{=}} [*] \oplus M_b \oplus ([c - \ell_b] \cap \neg([*] \oplus M_b \oplus [*]))$	$(s_a[x + \ell_b : x] = s_b) \wedge \left(\bigvee_{x < i \leq \ell_a - \ell_b} s_a[i + \ell_b : i] \neq s_b\right)$
	$M_b \mathrel{:\overset{\cap}{=}} \mathit{substrings}(M_a, c, c + \ell_b)$	
(substring, s_a, n, s_x)	$M_a \mathrel{:\overset{\cap}{=}} [n] \oplus M_x \oplus [*]$	$s_a[n + \ell_x : n] = s_x$
	$M_x \mathrel{:\overset{\cap}{=}} \mathit{substrings}(M_a, n, \infty)$	
(substring, s_a, n, k, s_x)	$M_a \mathrel{:\overset{\cap}{=}} [n] \oplus M_x \oplus [*]$	$s_a[n + \ell_x : n] = s_x$
	$M_x \mathrel{:\overset{\cap}{=}} \mathit{substrings}(M_a, n, k)$	
(trim, s_a, s_x)	$M_a \mathrel{:\overset{\cap}{=}} [␣, *] \oplus M_x \oplus [␣, *]$	$(s_x[0] \neq ␣) \wedge (s_x[\ell_x - 1] \neq ␣)$
	$M_b \mathrel{:\overset{\cap}{=}} \mathit{trim}(M_a)$	$\wedge \bigvee_{0 \leq i < \ell_a - \ell_x} \left(\begin{array}{ll} & \bigwedge_{0 \leq j < i} s_a[j] = ␣ \\ \wedge & s_a[i + \ell_x : i] = s_x \\ \wedge & \bigwedge_{i + \ell_x \leq j < \ell_a} s_a[j] = ␣ \end{array} \right)$
(contains, s_a, s_b)	$M_a \mathrel{:\overset{\cap}{=}} [*] \oplus M_b \oplus [*]$	$\bigvee_{0 \leq i < \ell_a - \ell_b} s_a[i + \ell_b : i] = s_b$
	$M_b \mathrel{:\overset{\cap}{=}} \mathit{substrings}(M_a, 0, \infty)$	
(endsWith, s_a, s_b)	$M_a \mathrel{:\overset{\cap}{=}} [*] \oplus M_b$	$s_a[\ell_a : \ell_a - \ell_b] = s_b$
	$M_b \mathrel{:\overset{\cap}{=}} \mathit{suffixes}(M_a)$	
(startsWith, s_a, s_b)	$M_a \mathrel{:\overset{\cap}{=}} M_b \oplus [*]$	$s_a[\ell_b : 0] = s_b$
	$M_b \mathrel{:\overset{\cap}{=}} \mathit{prefixes}(M_a)$	

```
CHECKNEGATIVEEDGES($PathCondition\ pc$,
        $string\ graph\ sg, \mathcal{M} = (M_1, M_2, \ldots, M_k)$)
1  $S_1 \leftarrow sol(M_1, k)$, $S_2 \leftarrow sol(M_2, k)$, $\ldots$, $S_k \leftarrow sol(M_k, k)$
2  $S_T \leftarrow S_1 \times S_2 \times \ldots \times S_k$
3  for $t$ in $S_T$:
4      $s_1 \leftarrow t_1$, $s_2 \leftarrow t_2$, $\ldots$, $s_k \leftarrow t_k$
5      if $verify(sg)$:
6          return $t$
7  return $\emptyset$
```

Figure 3.13: Solving of negative edges during automaton based solving

As an example, given the constraint:

$$\begin{array}{l}(\mathsf{notEquals},\ s_1,\ s_2) \wedge \\ (\mathsf{notEquals},\ s_2,\ s_3) \wedge \\ (\mathsf{notEquals},\ s_3,\ s_1)\end{array}$$

And given we know that the possible solutions for each string variable are:

Variable (v)	$sol(v, 3)$
$\mathtt{sv}_1$	{aaa, aab, aac}
$\mathtt{sv}_2$	{aaa, aab, aac}
$\mathtt{sv}_3$	{aaa, aab, aac}

Giving the cross product (S_T), given in Figure 3.14

{{aaa, aaa, aaa},}
{{aaa, aaa, aab},}
{{aaa, aaa, aac},}
{{aaa, aab, aaa},}
{{aaa, aab, aab},}
{{aaa, aab, aac},}
.
.
.
{{aac, aac, aac},}

Figure 3.14: Cross product S_T

Each tuple t within S_T will be taken and applied on the string graph, by assigning each element, t_i, to its respective vertex, s_i. Thereafter, *verify* will be used to check if the assignment satisfies the string graph.

In this example, when the tuple t {`aaa`, `aab`, `aac`} (or any other tuple where each element is unique) is selected, *verify* will return with a `true` result, and the solutions will be assigned to their respective symbolic string variables.

3.6.2 Translation to bitvectors

The bitvector approach to string constraints is in some sense more straightforward than that of automata, but involves difficulties of its own. Each of the hyperedges is translated as a constraint, the constraints are conjoined and then passed to a constraint solver. Unfortunately, the translation requires a fixed length for each of the vertices. Since these lengths are not known *a priori*, the translation process and the invocation of the solver need to be repeated for all possible lengths, up to a preset bound. Clever use of heuristics can limit the search space, but there are cases that are beyond the scope of this technique.

Consider, again, the hyperedge $(\textsf{contains}, s_a, s_b)$. Suppose that the current estimates for the lengths of s_a and s_b are 5 and 3, respectively. The resulting constraint for this hyperedge is

$$\begin{array}{ll} & (s_a[0] = s_b[0] \wedge s_a[1] = s_b[1] \wedge s_a[2] = s_b[2]) \\ \vee & (s_a[1] = s_b[0] \wedge s_a[2] = s_b[1] \wedge s_a[3] = s_b[2]) \\ \vee & (s_a[2] = s_b[0] \wedge s_a[3] = s_b[1] \wedge s_a[4] = s_b[2]) \end{array}$$

If, as in our case, the underlying constraint solver supports arrays, this can be simplified to

$$s_a[0:2] = s_b \vee s_a[1:3] = s_b \vee s_a[2:4] = s_b$$

The translation of string graph hyperedges to bitvector constraints are shown in the right-hand column of Table 3.2.

The bitvector solver's algorithm is given in Figure 3.15. The function $\textit{declare}(v, l)$ defines an appropriate bitvector variable with length l. The function $\textit{translate}(e)$ translates the edge e into the respective constraints (given in Table 3.2) and $\textit{push}(C)$ pushes the constraint C onto the stack of the bitvector

```
BITVECTORSOLVER($PathCondition\ pc$,
        $string\ graph\ sg = (I \cup S, E, \Omega)$)
 1  for string vertex $s_i \in S$:
 2      $declare(s_i, l_i)$
 3  for $e$ in $E$
 4      for $s_i$ connected to $e$:
 5          $C \leftarrow translate(e)$
 6          $r \leftarrow push(C)$
 7          if $r =$ false :
 8              $pc \leftarrow pc \cup$ FRESHCONSTRAINTS$(pc, sg)$
 9              return (false, $pc$, $sg$)
10  return (true, $pc$, $sg$)
```

Figure 3.15: The bitvector-based solver algorithm

solver which returns either **true** or **false**.

Reading the algorithm in Figure 3.15 shows that a representative bitvector variable is declared for each string variable. With every edge the appropriate constraints are pushed onto the bitvector constraint stack. This gives a way of detecting as early as possible if the string graph is unsatisfiable. An alternative method would have been to translate the entire string graph and then pushing all constraints in one big push.

Note that the algorithm in Figure 3.15 is actually quite similar to the algorithm in Figure 3.12. Both need to setup their variables (line 2 in both algorithms), both iterate over each edge, although the bitvector approach always iterates over each edge once whereas automaton may have to iterate over an edge many times. Also, both take the same action if an edge is found to be unsatisfiable (line $7 - 9$ in Figure 3.15 and lines $8 - 10$ in Figure 3.12).

3.7 Interchange

What has been omitted up to this point is an explanation of the function FRESHCONSTRIANTS(pc, sg). This section explains why it is needed and how it works.

As explained earlier, any symbolic integer variable in the *string graph*, is solved before reaching the string solver. This leads the string solver to see any symbolic integer variable as a constant. These constants are also referred to as best guesses.

During the string solver's attempt at solving the string graph, it may find that there is no satisfying solution. If this is the case, the string solver will conclude the integer solutions are incorrect and will add a disjunction to exclude each of the integer solutions. To clarify this, let us give an example. Consider a problem with two symbolic strings a and b, with lengths l_a and l_b guessed as 5 and 3, respectively. When the string solver concludes that values of a and b with these lengths do not satisfy the problem, it adds the following constraint to the integer constraints:

$$l_a \neq 5 \vee l_b \neq 3$$

After these new integer constraints are added, the string solver returns UNSAT. If the string solver terminates, with a change to the integer constraints, the loop on line 4 in Figure 3.7 is repeated. If the string solver terminates with no addition to the set of integer constraints, we conclude that the problem is satisfiable or not, depending on the output of the string solver.

When using the automata approach, queries can be made about the state of string variables to help the integer solver. For instance, during the string constraint solving, if the constraint

$$s_1.\texttt{indexOf("ab")} = 2$$

is encountered then s_1 must at least contain "`ab`". Thus, we can query the automaton representing s_1 to determine whether it contains the language `.*ab.*`. If it is found not to contain the language, we can add the integer constraint:

$$s_1.\texttt{indexOf("ab")} = -1$$

Various integer constraints can be derived from string constraints during solving, very similar to the preprocessor and its deriving of integer constraints, but only when using the automata approach. This is because when using the automata approach it stores the states of each symbolic string as they are solved. With almost no overhead we can query the state of these symbolic strings. This could also be done with the bitvector approach, but this entails a building up of temporary constraints, calling the SMT solver, then back-tracking the SMT solver to a state before it was made aware of the temporary constraints. In our experience keeping the SMT-solver calls at a minimum is best.

These new integer constraints that are learned during the solving process are only added after the current string solving iteration is complete and no satisfiable solution was founded.

3.8 Running Example

Following is an illustration of our entire approach. It starts with a Java program, deduces the path conditions, builds and preprocesses of a string graph, translates it to automaton operations and bitvector constraints, solves it, and finally performs the interchange needed. For this example we will restrict the possible characters to only the set {␣, `a`, `b`}

Figure 3.16 is a Java program consisting of one function with two string parameters. On line 1 there is a possible branch if the first string parameter start with a space and if the parameters are equal. Line 2 creates a new string variable by stripping off any leading or trailing spaces. (It is interesting to note that this new variable will never be able to satisfy the `if` condition on line 1.) Line 3 branches if our new variable ends with the string '`ab`'; otherwise if the two parameters are unequal. Lastly the last possible branch is on line 9 if the first string parameter has the substring '`a`' starting at index 5.

This example Java program is artificial and serves no purpose, except to explain our work. Given any two random strings it will be difficult to exercise any of the four branches, and thus our work should be a good candidate to exercise this program and all of its possible branches.

In order to exercise these branches we first need to construct all possible path conditions. In this example there are 12 path conditions, but we only select two for this example.

```
   public static void SOME_FUNCTION(String s1, String s2) {
1     if (s1.startsWith(' ') ∧ s1.equals(s2)) {
2        String newS1 = s1.trim();
3        if (newS1.endsWith('ab')) {
4           //Do Something
5        } else if (¬ s1.equals(s2)) {
6           //Do Something else
7        }
8     }
9     if (s1.indexOf(5).equals('a')) {
10       //Do Another thing
11    }
   }
```

Figure 3.16: An example Java program

The first path condition takes the line 1 branch, followed by the branch on line 5, and finally the branch on line 9. In order to satisfy this possible flow of data the following constraint must be satisfied:

Constraints		**Edges**
s1.**startsWith**(' ')	$\wedge$	(startsWith, s_1, $c_{␣}$)
s1.**equals**(*s2*)	$\wedge$	(equals, s_1, s_2)
$\neg$ *newS1*.**endsWith**('ab')	$\wedge$	(notEndsWith, s_3, c_{ab})
$\neg$ *s1*.**equals**(*s2*)	$\wedge$	(notEquals, s_1, s_2)
s1.**indexOf**(5) == ('a')		(indexOf, s_1, c_{a}, 5)
		(trim, s_1, s_3)

Choosing this path condition leads us to construct the string graph given in Figure 3.17 with the hyperedges given in the above table.

Running the preprocessor over the given graph will reveal that it is definitely unsatisfiable due to the pair of edges (equals, s_1, s_2) and (notEquals, s_1, s_2). The preprocessor will conclude with UNSAT and a new path condition will be chosen.

Assume the next path condition chosen is in the case where the branches on line 1, line 3 and line 9 are taken. In other words, *s1* must start with the space character, must be equal to *s2*, after being trimmed must end with the

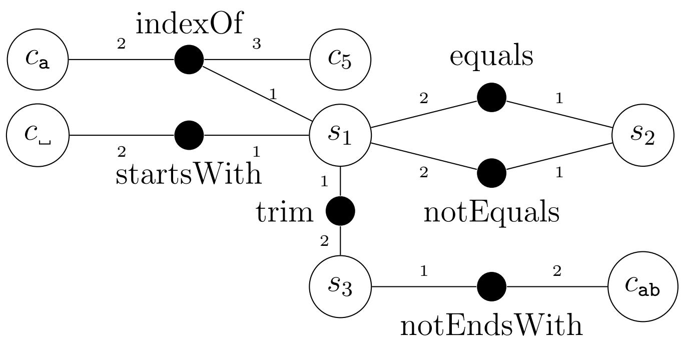


Figure 3.17: String graph 1

string ab and have the character a at the fifth index (and never before the fifth index). This produces the constraints:

Constraints		Edges
s1.**startsWith**(' ')	$\wedge$	(startsWith, s_1, $c_␣$)
s1.**equals**(*s2*)	$\wedge$	(equals, s_1, s_2)
newS1.**endsWith**('ab')	$\wedge$	(endsWith, s_3, c_{ab})
s1.**indexOf**(5).**equals**('a')		(indexOf, s_1, c_a, 5)
		(trim, s_1, s_3)

This new path condition will lead to the string graph in Figure 3.18.

Applying our preprocessor to this new string graph removes vertex s_2 from the string graph (keeping in mind to map s_1's solution to s_2's solution). When the solutions are reported back to the callee, the necessary reverse mapping occurs leaving the callee unaware of the stripping of vertices. The following integer constraints will be added in order to prepare the integer constraint

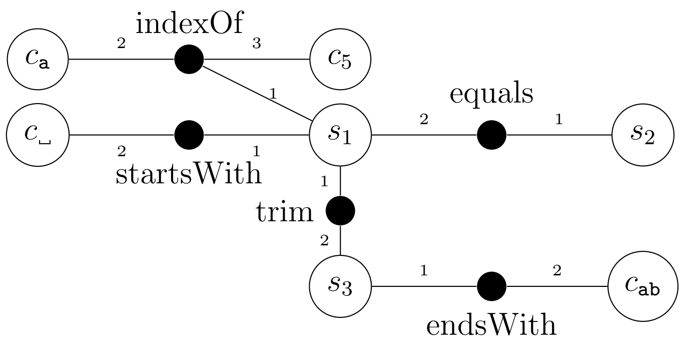


Figure 3.18: String graph 2

solver for its first set of solutions or, as our approach calls it, best guesses (l_i is the length of the symbolic string represented by s_i):

Constraint		**Reason**
$l_1 > 0$	$\wedge$	
$l_2 > 0$	$\wedge$	
$l_3 > 0$	$\wedge$	
$l_1 = l_2$	$\wedge$	
$l_3 \geq 2$	$\wedge$	s_3 must be able to contain at least `ab`
$l_1 \geq l_3$	$\wedge$	The trimmed version of s_1 must be equal or smaller than itself
$l_1 \geq 1$	$\wedge$	s_1 contain at least one space
$l_1 > 5$		s_1 must have index 5 for it to be able to place `a`

Before starting the string constraint solver, the integer solver is invoked to give its first best guess, assume these values are: $l_1 = 6, l_2 = 6, l_3 = 2$. These values are a reasonable assumption as they are the smallest values for which each variable will satisfy the integer constraint.

Consider the automata approach first. Map each symbolic string s_i to its own automaton M_i. Each M_i automaton is initialised to contain all words of length l_i. Thus $M_1 = ...$ and $M_3 = ..$ Stepping through the edges:

1. (startsWith, s_1, $c_␣$): M_1 is intersected with the language `␣.*` and assigned back to M_1, resulting in the language `␣.....` for M_1

2. (trim, s_1, s_3): M_1 is intersected with the language of M_3 where each word has leading and trailing spaces ($M_1 \cap$ ␣* $\oplus$ M_3 $\oplus$ ␣*) resulting in the language `␣.....` for M_1. M_3 is intersected with the language of M_1 after stripping off leading and trailing spaces from each word in M_1, resulting in an M_3 with language `[a-b][a-b]`

3. (indexOf, s_1, $c_{\mathtt{a}}, c_5$): M_1 is intersected with the language `.....a.*` resulting in $M_1 =$ `␣....a`

4. (endsWith, s_3, $c_{\mathtt{ab}}$): M_3 is intersected with the language `ab` resulting in $M_3 =$ `ab`

In our approach we iterate over each edge a second time because each edge has caused a change in one or more automata and has therefore been placed in the work set (line 12 of Figure 3.12). For brevity's sake we jump ahead to the only edge of interest. Once we come across (trim, s_1, s_3) where $M_1 =$ `␣....a` and $M_3 =$ `ab` we will find that intersections are empty. This is due to the fact that M_1's words all end with `a` and M_3's only word ends with `b`.

This leads the string solver to return UNSAT, and fresh integer constraints are added to those originally used to prepare the integer constraint solver. The only integer constraints added are $l_1 \neq 6 \vee l_3 \neq 2$. There may be some back and forth between the two solvers as incorrect best guesses are tried. We will jump to one possible best guess which will work $l_1 = 7$, $l_2 = 7$ and $l_3 = 2$. After stepping through each edge once again, we end with $M_1 =$ `␣....a.` and $M_3 =$ `ab`. Now, (trim, s_1, s_3) edge will not lead to an empty intersection but will in fact modify M_1 to have the language `␣....ab`.

Selecting the shortest possible solutions from the two automata will lead us to assign `␣␣␣␣␣ab` to s_1 (and also to s_2 due to us linking their two solutions), and `ab` to s_3.

Moving on to the bitvector approach, we return to our best guesses as $l_1 = 6$, $l_2 = 6$ and $l_3 = 2$. Two bitvector variables are initialised b_1 and b_3 with lengths 6 and 2, respectively. Stepping through each edge:

1. (startsWith, s_1, $c_{␣}$): the constraint b_1 [0] = `␣` is added.

2. (trim, s_1, s_3): several constraints are added of the form of b_1 $[i] = b_3$ [0] $\wedge$ b_1 $[i+1] = b_3$ [1], where i iterates over the length of b_1. Additionally b_3 [0] $\neq$ `␣` $\wedge$ b_3 [1] $\neq$ `␣`.

3. (indexOf, s_1, $c_{\texttt{a}}$, c_5): the constraint b_1 [5] = `a` is added. To ensure `a` is at least at index 5, add the constraints: b_1 [0] $\neq$ `a` $\wedge$ b_1 [1] $\neq$ `a` $\wedge$ $b_1[2] \neq$ `a` $\wedge$ $b_1[3] \neq$ `a` $\wedge$ b_1 [4] $\neq$ `a`.

4. (endsWith, s_3, $c_{\texttt{ab}}$): the constraints b_3 [0] = `a` $\wedge$ b_3 [1] = `b` is added.

After each edge we push the constraints onto the bitvector solver's stack, with it reporting SAT or UNSAT. When the constraints for endsWith are pushed it returns UNSAT due to b_1 ending with the character `a` and b_3 ending with the character `b`. Once UNSAT has been received from the bitvector solver, we stop translating and add our fresh integer constraints, which in this case will be: $l_1 \neq 6 \vee l_3 \neq 2$. The SMT solver's state is cleared and the translation will start from the first edge again. Several best guesses and translation may be made.

We will skip the dead-end best guesses and consider the case where the best guesses $l_1 = 7$, $l_2 = 7$ and $l_3 = 2$ are returned. This time around the translation would be very similar to previous iteration except for the fact that b_1 can now end with the character `b`, leading our bitvector solver to report SAT and returning satisfiable assignments. These assignments could be of the form $s_1 \leftarrow$ `␣␣␣␣␣ab` and $s_3 \leftarrow$ `ab`.

3.9 Comparison to other work

Once again we dedicate a section to a comparison to other published work.

3.9.1 Hooimejier's Lazy approach

Our approach also constructs a graph, but ours tend to be much more verbose. Intermediate edges and variables are created in an attempt to break down certain constraints into simpler steps, e.g., the `trim` operation causes our string graph to create an intermediate vertex representing the trimmed variable. Unlike Hooimeijer, we try to simplify and solve the graph by simple graph algorithms before trying to solve it.

Hooimeijer's graph is accompanied by a mapping between each string constraint and what edges and vertices are involved with that constraint. Our graph does not need such a mapping.

The graph exploration of Hooimeijer is a much more involved process. Selecting a certain few solutions with a guarantee that they are at least somewhat good is tricky, and backtracking unnecessarily is difficult to avoid. Our approach is a much simpler process of walking through the edges until the automata convergence. Hooimeijer's approach can find a solution much faster with less memory usage, but only if it is very lucky and makes exactly the right choices while processing his string graph.

Like our approach it uses automata to calculate solutions, although it appears that Hooimeijer discards all calculated automata after selecting a solution, which could lead to penalties when backtracking.

3.9.2 HAMPI

HAMPI is only concerned with solving string constraints, not with symbolic execution or the constructing of string constraints, although it has had success when integrated with other tools.

Compared to our approach, HAMPI also uses intermediate representation, although theirs is a language expressed in a brief grammar which is able to express all the string constraints they are able to solve.

It lacks support for symbolic integers which means it does not support `charAt` and `indexOf` operations. It does however support regular expressions and grammars. HAMPI uses the STP SMT-Solver, with memoization to minimise its work.

Our approach does have memoization due to JPF-symbc storing our internal state with every path condition solved. Therefore, when a constraint is added to an existing path condition, we will solve that path condition with some of our previous state restored.

3.9.3 Kaluza

Kaluza acts as a layer on top of HAMPI, it adds features to compensate for limitation of HAMPI. One of these features is a graph to keep track of how string variables are dependant on each other. This is especially valuable on constraints such as `contains` or `concat`.

Kudzu, as a whole, does not support symbolic integer-string operations, for example, `charAt`, `indexOf` and `substring`. It does, however support regular expressions, replace and split. Even though we have not implemented them, our approach (a string graph with either automata, bitvectors, or both) could be extended to support these operations without much difficulty.

3.9.4 JSA

Compared to us, the approach of Christensen et al. is capable of handling the entire Java string API, but most of this understanding is approximate. Also, Christensen et al. is capable of working over the entire Unicode, whereas we restrict ourself to the 32 - 127 range of the ANSI code.

It seems Christensen et al. is only able to consider one symbolic string variable. Also, their approach does not consider symbolic integer input.

Our automata approach is built on top of the library they built for their approach.

3.10 Overview

Java programs are executed within a virtual machine which is aware of symbolic execution. This enables the virtual machine to build a series of path conditions which in turn is passed to our constraint solver. Given a path condition our constraint solver builds a string graph. Any symbolic integer is approximated with integer constants. The string graph is propagated with these integer constants and then solved either by automata operations or bitvector constraints. If the solver returns UNSAT than all other possible integer constant approximations are exercised until either the problem is found to be satisfiable or there are no integer constant approximations.

With the tool now implemented, the next chapter will cover how it performs on string path conditions, and how the performance of the automata approach and bitvector approach compares.

Chapter 4

Results

In order to evaluate our work, we have implemented the approach given in Section 3.1 in Java, by extending the JPF-symbc plugin for JPF-core. We have added a time-out mechanism in order to help scale to larger programs.

Because the choice of which string solver to use is delayed as long as possible, we believe both approaches have an equal platform from which to perform. This platform consists of an already non-trivially unsatisfiable, simplified string graph and the same initial best guess on possible values for all the symbolic integers.

First we will be applying our approach to some real-world examples, some of which have been used by other related work, and also one new example which is much larger than what previous work has used. Thereafter, we will evaluate our approach with randomly generated input, in order to get a sense of performance differences between the automata and bitvector approaches.

4.1 Real-world

We have selected three real-world examples to evaluate our work with. Two of these examples have already been cited by other related work, and the third is a new example we introduce. The two already in existence is the **EasyChair** (21 LOC) and **WU_FTPD** (20 LOC) examples, while **Mystery** (311 LOC) is the real-world example that we are introducing. Unlike **EasyChair** and **WU_FTPD**, **Mystery** consists of many more lines of code and even more path conditions.

WU_FTPD has been discussed in Section 1.1. **EasyChair** is a URL sanity check. It verifies that a URL string (1) starts with 'http://', (2) is directed toward Microsoft Live Search or Google and (3) it is addressing the 'EasyChair' query. Unfortunately the source of **Mystery** may not be disclosed, but its nature can be. It is intended to take some HTML page as a string, remove certain tags from the page and return the stripped down version. It consists of 311 lines of code, and once it was deployed in the field, it fell into an infinite loop. Although we cannot detect infinite loops (due to the Halting Problem, [30]) we were able to identify an assumption that was made during the coding of the software, and were able to to see whether that assumption holds throughout the run of the software. The programmer of the original source code decided to use an integer i to denote the character index within the string, and once i has reached the last character the program stops. i was assumed to be always increasing, which seems simple, but because the programmer used Java String API calls to jump ahead in the string the person forgot that those calls could return -1 effectively resetting i back to the starting position of the string. It was this assumption which did not hold, and can be detected with our software.

Each real-world example was run through our implementation, with both the automata and bitvectors approach. The time spent in string constraint solving and integer constraint solving was measured. Also the number of times the string solver needed to iterate due to bad guesses from the integer constraint solver was measured. To be able to scale we found setting a 3 second time limit per path condition worked well. With some investigation we found that if the string solvers tries to solve a path condition for longer then 3 seconds, then that path condition is most likely to have been found unsatisfiable if enough time was given. Thus we were able to trim off unsatisfiable paths from the input program quicker than without a time limit.

The experiments were performed on a two-core Intel Core2Duo CPU with each core clocked at 2.00Ghz. 2GB of DDR2 memory was available to the operating system: Ubuntu 11.10 64-bit running on Linux 3.0.0.

Although most of these programs contain real bugs which we are able to

Table 4.1: Results for real-world inputs, traversing all paths; all times in milliseconds

Model	String	Integer	Iterate	PCs	Preprocessed	Timeouts
			Automata			
WU_FTPD	192	0	0	7	0	3
EasyChair	4668	756	21	15	1	0
Mystery	643123	2526	358	6148	1628	1258
			Bitvectors			
WU_FTPD	122	0	0	7	0	2
EasyChair	1066	127	21	15	1	0
Mystery	113299	673	109	6148	1628	916

identify, we chose to ignore them at first. This is because of the time taken to find a bug within the program is directly related to the search heuristic used to step through each possible path condition. To this end we simply measured the time taken to step through all possible path conditions, thus producing results that more accurately reflect the solver in use, rather than the search heuristic.

Another real-world consideration we had to make is to limit the program's state depth. This was due to **Mystery** containing infinite loops, and thus, infinite depth. The depth limit was set to 33 states. The depth limit was determined by the minimum number needed to find the infinite loop bug.

Our results are given in Table 4.1. The `String` column is the total time taken, in milliseconds, to solve the string constraints. `Integer` is the total time taken, in milliseconds, to solve the integer constraints. `Iterate` indicates the total number of times interchange was needed between our string solving and integer solving. `PCs` is the total number of path conditions used in calculating the solving averages. These first four columns are only calculated on the path conditions **that did not time out on either approach**; the reason for this is motivated later. `Preprocessed` is the amount of path conditions that were found definitely unsatisfiable by the preprocessor. `Timeout` is the number which was not solved and declared UNSAT after our 3 second time-out.

The reason for comparing the times on only those path conditions which did not time out for either approach is because we wish to compare apples with apples. If we were to compare all path conditions from the one approach with all the other path conditions from a different approach we would be comparing two different sets of path conditions. The difference in the sets of path conditions is due to the time-out mechanism and incremental build-up of path conditions by JPF. Everytime JPF appends a constraint to a path condition, our symbolic string execution is invoked. If the symbolic string execution reports UNSAT, then JPF will stop appending and continue with a different branch of path conditions. If, however, the symbolic string execution reports SAT, JPF will continue appending the next constraint and the process repeats. Because we have a time-out mechanism, the different approaches will cause JPF to stop building various path conditions at different places giving a different set of path conditions for each approach. These sets will have some elements in common, and it is those elements we identify and measure with.

An unfortunate side-effect of comparing only those path conditions that did not time-out is that for **WU_FTPD** the one path condition which resulted in interesting behaviour was lost, because the automata approach timed out.

From Table 4.1 it is clear that the bitvector approach is superior. However, it needs to be stressed that each of these real-world examples only exercise a certain class of string graphs. There may be other classes of string graphs for which automata are superior. Therefore, the next section will be looking at random string graphs.

4.2 Randomised

In order to identify the unique performance characteristics of each approach, we chose to generate a random set of small path conditions. An invaluable side product of this was the identification of subtle program mistakes made during the implementing of the approach. The generation of tests was done at the string graph level, thus the constructing, preprocessing, translation and interchange were evaluated.

Although the performance for the majority of random problems are quite uniform, there are some extremes. Most of the problems did not make it past the preprocessor, which shows the effectiveness of the preprocessor, but also is

Table 4.2: Results for random inputs; all times in milliseconds

			Bitvectors				**Automata**			
	Edges	**PP Edges**	**Interchange**	**String**	**Integer**	**Total IC**	**Interchange**	**String**	**Integer**	**Total IC**
5 Edges SAT										
Average	4.99	4.18	3.14	41.55	294.03	14.52	2.57	39.77	270.69	13.95
Median	5	4	1	15	11	12	1	10	11	12
Min	3	0	1	0	3	3	1	0	3	3
Max	5	5	164	2900	60320	178	215	29134	111142	228
10 Edges SAT										
Average	9.93	8.10	9.36	158.49	1439.76	25.07	6.57	166.26	1162.80	22.29
Median	10	8	1	25.5	20	17	1	21	18	17
Min	9	1	1	10	4	7	1	4	5	7
Max	10	10	112	4243	41870	131	148	3828	70108	174
5 Edges UNSAT (Z3 Timeouts: 140, Automata Timeouts: 184)										
Average	4.99	4.45	23.62	380.09	3812.70	35.13	23.64	472.48	3334.31	35.15
Median	5	5	7	82	80	19	7	36	105.5	20
Min	4	2	1	8	5	8	2	1	5	8
Max	5	5	253	9369	105179	264	226	66424	98961	235
10 Edges UNSAT (Z3 Timeouts: 80, Automata Timeouts: 98)										
Average	9.94	8.41	31.06	662.84	7988.26	48.03	35.69	624.51	8509.76	52.66
Median	10	9	10.5	144.5	226.5	28	11	86	269	29
Min	9	5	2	7	11	14	2	1	9	13
Max	10	10	216	5641	113848	232	216	5685	113507	232
All										
Average	5.20	4.36	4.73	70.09	601.24	16.32	4.18	73.30	550.79	15.76
Median	5	4	1	15	12	13	1	10	11	13
Min	3	0	1	0	3	3	1	0	3	3
Max	10	10	253	9369	113848	264	226	66424	113507	235

a side-effect of random generated problems.

We will first investigate the overall performance by looking at Table 4.2. In order to generate the needed data for this table we randomly generated 30328 problems with 5 or 10 constraints each. The constraints were randomly selected from the set:

`charAt`	`endswith`	`notcontains`	`notstartswith`	`trim`
`concat`	`equals`	`notendswith`	`startswith`	
`contains`	`indexOf`	`notequals`	`substring`	

Only the first variation of indexOf and substring was considered (see Section 3.4). The number of symbolic variables was 6 at the most. Each problem was passed to our automata and bitvector approach with a time-out of 120 seconds.

In the given table, the first column shows the number of edges within the string graphs, the second column shows the number of edges after preprocessing completed. Then for each approach: `interchange` shows the number of times the solver iterated between the string solver and the integer solver, `string` indicates the total time spent in the string constraint solver, `integer` indicates the total time spent in the integer constraint solver and, lastly, `Total IC` indicates the total number of integer constraints encountered during solving.

Of all the problems, the preprocessor found 13398 unsatisfiable within a few milliseconds, which shows the effectiveness of the preprocessor. Of the 16930 problems that passed the preprocessor phase, 10139 where found UNSAT and 6619 SAT. Of those found UNSAT, 9398 were found to have no satisfiable solution for the integer constraints generated by the preprocessor, hence the string solver was never even invoked. Of all 30328 inputs, only 339 timed out, the reason for which will be discussed a little bit later.

The given table only uses those data points which passed the preprocessor and were able to generate at least one best guess.

If we were to compare the total averages of the automaton and bitvector approaches (73.30 versus 70.09 milliseconds) it is difficult to state that one method is dramatically better than the other. Although, comparing the number of time-outs, bitvectors perform better then automata, which would

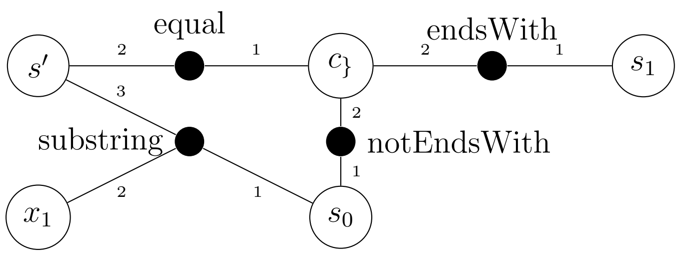


Figure 4.1: Extreme 1

indicate that the automata approach is more susceptible to weak performance when facing difficult constraints. The automata approach does, however, iterate fewer: 4.18 compared to 4.73 on average.

One problem that demonstrated an extreme difference between the two approaches is given in Figure 4.1. The graph demonstrates a few constraints applied to two symbolic input strings s_0 and s_1. (substring, s_0, s', x_1) shows a constraint where at some index x_1 (x_1 is a symbolic integer) the temporary symbolic string s' must start and stretch until the end of s_0. In this regard there is an implicit constraint placed on the graph which is that s_0 must be ending with s'. The temporary string s' must be equal to $\}$, s_0 must not end with $\}$ and s_1 must end with $\}$.

Executing this graph, the bitvector approach returns UNSAT within 1 second, but the automata approach ends up timing out after our 120 seconds time-out limit. Upon closer investigation we found the UNSAT resolve by the bitvector approach to be correct, because the string graph is forcing s_0 to end with $\}$ via the edges (substring, s_0, s', x_1) and (equal, s', $c_\}$), but, there is a conflicting edge (notEndsWith, s_0, $c_\}$), and thus the graph is unsatisfiable.

Now, the bitvector approach traverses this particular string graph in the order of substring, notEndsWith, endsWith. Remember the preprocessor would have removed the equals hyperedge and merged the s' and $c_\}$ vertices. Due to this lucky traversal the bitvector solver would not get past the subgraph consisting of the substring and notEndsWith hyperedge, and will conclude the entire graph to be unsatisfiable. Note that it never had to even consider the s_1 symbolic string.

The automata approach, however, would split the hyperedges into two sets, one containing the positive constraints: substring, endsWith and the other con-

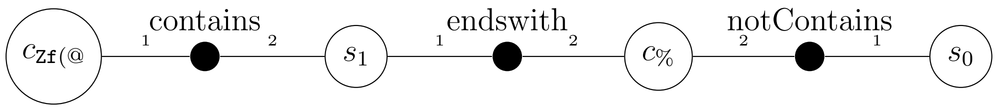


Figure 4.2: Extreme 2

taining the negative constraints: notEndsWith. Now because it will try and solve the positive constraints first and the negative constraints later, it will be encountering the unsatisfiable nature of the graph later than the bitvector approach did. Because the automata approach is now traversing the edges in an inefficient sequence, it would be considering s_1 during the interchange phase. Before, with the bitvector approach there was only two symbolic integers present in the interchange phase, l_0 and x_1, leading to 30×30 possible combinations. But because the automata approach is adding l_1 to the the interchange phase, we have $30 \times 30 \times 30$ possibilities. This exponential growth causes our integer constraint solver to struggle with constraints later on in the interchange phase, leading to long solving times, and the eventual time-out.

We do have the opposite extreme as well. Consider the constraints in Figure 4.2, which state that s_0 should not contain `%`, s_1 should end with `%` and that s_1 should be contained within `ZF(@`. This problem is unsatisfiable because s_1 cannot contain `%` and be a substring of `ZF(@`. It is important to note that notcontains edge is unnecessary in proving this graph unsatisfiable.

In this example automata performs well, and the bitvector struggles, although it does not time-out. There are two reasons why automata perform better then bitvector:

1. **Ordering of constraints**: once again automata postpone the negative constraints (notContains) untill the end, which in this case was a better decision then tackling it first as with the bitvector approach.

2. **In control of the current state of each symbolic string**: the automata approach keeps an automaton for each symbolic string, which means it can quickly enquire about the structure of each symbolic string. The bitvector approach has no such capability as its internal state is hidden from JPF-symbc.

The first case once again causes an exponential explosion in the interchange process which could have been avoided. The second case is interesting though. When dealing with automata it is clear that automaton `.*%` would cause an empty intersection with any substring of `Zf(@`, and thus the automaton can declare the path condition unsatisfiable quickly. The bitvector approach, however, is unable to reason as such, and simply starts to iterate through all the possible integer values which could be assigned to the symbolic integers, lengthening the process by a considerable amount.

4.3 Overview

We have applied our work to three real-world examples and several thousand random string graph instances. From the random instances it appears that bitvectors have a slight edge, although it is not as clear cut. If the outliers in the random results are ignored, the two approaches perform almost the same. An outlier is identified by results where the automata or bitvector solving time are different by 100-fold. In our data there are 93 outliers (or 1.4% of all data). After removing these outliers the average string constraint solving time for bitvectors is 3.72 milliseconds and the 3.99 milliseconds for automata. These averages are so close it is difficult to claim one as superior to the other.

In the real-world examples the bitvector approach is clearly the faster one. Unfortunately, as of yet, we are unable to determine exactly what structure in these examples are giving the automata approach a hard time.

Chapter 5

Conclusion and Future work

We have created an approach and implementation which is comparable to other state of the art string constraint solvers and symbolic string execution engines. What makes our approach novel is the ability to switch between bitvectors and automata solving. This leads to the ability to compare the performance differences between the automata approach and the bitvector approach. From this, we were not able to see a significant difference between the two.

The automata approach is a lot more work to implement, and is thus prone to containing more implementation flaws than bitvectors. Also, it does not take to negative constraints naturally, which needs to be added on in addition to the automata theory. However, it does perform well when the constraints are small, or when it can exploit certain enquiries into the state of symbolic strings, leading to better integer constraints.

The bitvector approach is much simpler because it uses a third party solver. Unfortunately this leaves us incapable of optimising it for our specific set of constraints. Nevertheless it performs well when the problem starts to scale up, even when we cannot enquire into the specific states of a symbolic string.

The preprocessor however, has a huge impact due to it being able to identify certain classes of constraints within the string graph and simplify them. Therefore for some weaknesses that may arise in either the bitvector or the automata approach we can counter by developing a better preprocessor which would identify these weaknesses ahead of time and make the necessary modification to achieve better performance.

As seen from the results the order in which edges are processed is important. If the string graph contains an unsatisfiable subgraph then it is better to identify that subgraph by considering the minimum amount of edges. During our work we have not considered reordering the edges in any way when translating them, and we are unaware of any work that has done this in terms of string constraint solving.

Minimizing the number of interchanges needed would be ideal. Our approach could save a lot of time if the best guesses generated by the integer constraint solver are more relevant to what is needed to solve the graph. Unfortunately, we do not see an immediate way of combining the string constraint solver and integer constraint solver into one solver (that would not need interchanging), such as [32] has suggested.

A worrying gap in research in this area is the handling of Unicode. [4] is able to store and manipulate any character from Unicode but only at a superficial level. Unicode presents various problems, some of which are the fact that a character in Unicode may consist of a variable amount of bits, characters are context sensitive and a character may have more than one representation in its original language.

An avenue we do believe we can immediately start to explore is how to solve certain path conditions with the automata approach and to solve others with the bit-vector approach all in the same execution.

List of References

[1] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[2] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19th *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[3] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. Path feasibility analysis for string-manipulating programs. In *Proc 15th Intl Conf on Tools and Algorithms for the Construction and Analysis of Systems*, volume 5505 of *LNCS*, pages 307–321. Springer, March 2009.

[4] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc 10th Intl Symposium on Static Analysis*, volume 2694 of *LNCS*, pages 1–18. Springer, June 2003.

[5] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. Technical report, Microsoft, 2008.

[6] Xiang Fu and Chung-Chih Li. Modeling regular replacement for string constraint solving. In *Proc 2nd NASA Formal Methods Symposium*, pages 67–76. NASA, April 2010.

[7] Xiang Fu, Xin Lu, Boris Peltsverger, Shijun Chen, Kai Qian, and Lixin Tao. A static analysis framework for detecting SQL injection vulnerabili-

ties. In *Proc 31st Annual Intl Computer Software and Applications Conf*, pages 87–96. IEEE Computer Society, July 2007.

[8] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proc 19th Intl Conf on Computer Aided Verification*, volume 4590 of *LNCS*, pages 519–531. Springer, July 2007.

[9] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test generation through programming in udita. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 225–234, New York, NY, USA, 2010. ACM.

[10] Milos Gligoric, Vilas Jagannath, and Darko Marinov. Mutmut: Efficient exploration for mutation testing of multithreaded code. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 55–64, Washington, DC, USA, 2010. IEEE Computer Society.

[11] Patrice Godefroid, Adam Kieżun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proc 2008 ACM SIGPLAN Conf on Programming Language Design and Implementation*, pages 206–215. ACM, June 2008.

[12] Alex Groce and Willem Visser. Model checking java programs using structural heuristics. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 12–21, New York, NY, USA, 2002. ACM.

[13] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso. A classification of SQL-Injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, Arlington, VA, USA, March 2006.

[14] Pieter Hooimeijer and Westley Weimer. A decision procedure for subset constraints over regular languages. In *Proc 2009 ACM SIGPLAN Conf on Programming Language Design and Implementation*, pages 188–198. ACM, June 2009.

[15] Pieter Hooimeijer and Westley Weimer. Solving string constraints lazily. In *Proc 25th IEEE/ACM Intl Conf on Automated Software Engineering*, pages 377–386. ACM, September 2010.

[16] Karthick Jayaraman, David Harvison, Vijay Ganesh, and Adam Kiezun. Jfuzz. a concolic whitebox fuzzer for java. In *Proceedings of the First NASA Formal Methods Symposium*, pages 121–125. Nasa, 2009.

[17] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proc 9th Intl Conf on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *LNCS*, pages 553–568. Springer, April 2003.

[18] Adam Kieżun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: a solver for string constraints. In *Proc 18th ACM/SIGSOFT Intl Symposium on Software Testing and Analysis*, pages 105–116. ACM, July 2009.

[19] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[20] Steven Lauterburg, Mirco Dotta, Darko Marinov, and Gul Agha. A framework for state-space exploration of java-based actor programs. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 468–479, Washington, DC, USA, 2009. IEEE Computer Society.

[21] Mark-Jan Nederhof. A general technique to train language models on language models. *Computational Linguistics*, 31, 2005.

[22] NIST. The economic impacts of inadequate infrastructure for software testing. Technical report, RTI, Health, Social and Economics Research, 2002.

[23] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. Differential symbolic execution. In *Proceedings of the 16th*

ACM SIGSOFT International Symposium on Foundations of software engineering, SIGSOFT '08/FSE-16, pages 226–237, New York, NY, USA, 2008. ACM.

[24] Corina Păsăreanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *Intl Jnl on Software Tools for Technology Transfer*, 11(4):339–353, October 2009.

[25] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael R. Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *Proc 17th ACM/SIGSOFT Intl Symposium on Software Testing and Analysis*, pages 15–26. ACM, July 2008.

[26] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for JavaScript. In *Proc 31st IEEE Symposium on Security and Privacy*, pages 513–528. IEEE Computer Society, May 2010.

[27] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for C. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[28] Daryl Shannon, Indradeep Ghosh, Sreeranga P. Rajan, and Sarfraz Khurshid. Efficient symbolic execution of strings for validating web applications. In *Proc 2nd Intl Workshop on Defects in Large Software Systems*, pages 22–26. ACM, July 2009.

[29] Daryl Shannon, Sukant Hajra, Alison Lee, Daiqian Zhan, and Sarfraz Khurshid. Abstracting symbolic execution with string analysis. In *Proc Testing: Academic and Industrial Conf, Practice and Research Techniques*, pages 13–22. IEEE Computer Society, July 2007.

[30] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, 2nd edition, 2005.

[31] Nikolai Tillmann and Jonathan de Halleux. Pex–white box test generation for .NET. In *Proc 2nd Intl Conf on Tests and Proofs*, volume 4966 of *LNCS*, pages 134–153. Springer, April 2008.

[32] Margus Veanes, Peli de Halleux, and Nikolai Tillmann. Rex: Symbolic regular expression explorer. In *Proc 3rd Intl Conf on Software Testing, Verification and Validation*, pages 498–507. IEEE Computer Society, April 2010.

[33] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering Jnl*, 10(2):203–232, April 2003.

[34] Fan Yu, Tevfik Bultan, and Oscar H. Ibarra. Relational string verification using multi-track automata. In *Proc 15th Intl Conf on Implementation and Application of Automata*, volume 6482 of *LNCS*, pages XXXXX–XXXXX. Springer, August 2010.