VIETNAM NATIONAL UNIVERSITY, HANOI

UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Tran Duc Muoi

SOME IMPROVEMENTS FOR REGULAR EXPRESSION SOLVING IN Z3 - REGEX

Major: Computer Science

HA NOI - 2015

VIETNAM NATIONAL UNIVERSITY, HANOI

UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Tran Duc Muoi

SOME IMPROVEMENTS FOR REGULAR EXPRESSION SOLVING IN Z3 - REGEX

Major: Computer Science

Supervisor: A/Prof. Truong Anh Hoang

HANOI - 2015

AUTHORSHIP

“I hereby declare that the work contained in this thesis is of my own and has not been previously submitted for a degree or diploma at this or any other higher education institution. To the best of my knowledge and belief, the thesis contains no materials previously published or written by another person except where due reference or acknowledgement is made.”

Signature:...

SUPPERVISOR’S APPROVAL

“I hereby approve that the thesis in its current form is ready for committee examination as a requirement for the Bachelor of Computer Science degree at the University of Engineering and Technology.”

Signature:..

ACKNOWLEDGMENTS

I would like to acknowledge the following people for their support and assistance in my final thesis.

First of all, I would like to give a special thankfulness to my supervisor, Associate Professor Truong Anh Hoang, Department of Software and Engineering, Faculty of Information Technology, University of Engineering and Technology, for giving me the instructions and explanations of several problems I have faced related to this thesis. I could not complete my thesis without a dozen of helpful advices from him in weekly meetings and via emails. Throughout two years I have researched under his guidance, I have learnt so much from him, both the academic works and the proper lifestyles as well.

I also want to thank Dinh Trung Anh, University of Engineering and Technology, Vietnam National University of Hanoi, who is my classmate at class K56CA, my teammate at the scientific research contest and my dearest companion in every event we participated together. He also introduced me to string solvers problems and had researched with me on this subject till last March. My thesis could not be done without him.

Since I am new to solvers, I have a lot of questions and my seniors have given me not only clear answers but also many advice and suggestions. Thank you so much for all your help, especially Trinh Minh Thai and Loi Luu.

Last but not least, I do really appreciate the supports from my family and all of my friends during my thesis time. Their supports did inspire and motivate me much whenever I went down. Honestly, my thesis as well as my life would have gone nowhere without any of you.

Ha Noi, May 2015

Tran Duc Muoi

ABSTRACT

In this thesis, we will explain Z3-regex[1], a string solver that support regular expression constraints. Z3-regex is built on top of Z3-str[2] [1]. Z3-str was introduced as the first SMT-based string solver that extends Z3 solver.

String solvers have many applications. They can be used for symbolic, static and dynamic analysis of web applications, which usually take string as the inputs. Since Z3-str does not handle constraints containing regular expressions, which normally define a set of string, so we extend it to support regular expressions.

Solving string constraints containing the regular expression currently is a hot research topic. The leading solver in the world now is S3[3] and we found it still has some deficiencies that motivated us to develop our tool. At the time we wrote this thesis, Z3-str developers have updated their tool to Z3-str2. Unfortunately, this version was not ready for us to redo the evaluation.

Since the first time Z3-regex was introduced [2], it has been improved technically in regular expression solving, especially in $Star()$ function. Z3-regex now can solve more cases. When I evaluated Z3-regex to compare to the state-of-the-art tool such as S3, it can solve several cases that S3 cannot.

Keywords: *SMT Solver, String Constraints, Regular Expression*

[1]`https://github.com/anhtrung93/Z3-str-Regex`
[2]`https://sites.google.com/site/z3strsolver/`
[3]`http://www.comp.nus.edu.sg/~trinhmt/S3/`

TÓM TẮT

Trong khóa luận này, chúng tôi sẽ giới thiệu Z3-regex[4] là một máy giải xâu (string solver) hỗ trợ giải các ràng buộc (constraint) của biểu thức chính quy (Regular Expression). Z3-regex được xây dựng dựa trên nền tảng có sẵn của Z3-str[5] [1]. Trong đó, Z3-str được xem là máy giải xâu đầu tiên dựa trên bộ giải Z3 SMT.

Một máy giải xâu có rất nhiều ứng dụng, ví dụ như dùng để biểu tượng hóa(symbolic), thống kê và phân tích động các ứng dụng web mà thông thường rất hay sử dụng các giá trị đầu vào là xâu. Do Z3-str không thể giải các ràng buộc liên quan đến các biểu thức chính quy, thông thường biểu thức chính quy lại định nghĩa một lớp các xâu, cho nên chúng tôi đã mở rộng nó để nó có thể hỗ trợ biểu thức chính quy.

Các vấn đề liên quan đến giải các ràng buộc xâu hiện nay đang là một chủ đề nghiên cứu rất nóng hổi và nhận được nhiều sự quan tâm của nhiều nhóm phát triển trên thế giới. Một trong những bộ giải xâu hàng đầu vào thời điểm hiện tại là bộ giải S3 [6]. Tuy nhiên, sau một thời gian nghiên cứu, chúng tôi phát hiện ra nó vẫn còn tồn tại một số điểm có thể cải tiến được. Điều này đã thúc đẩy chúng tôi phát triển công cụ giải xâu cho riêng mình. Vào thời gian khóa luận này được viết, nhóm phát triển Z3-str đã nâng cấp nó lên phiên bản Z3-str2 với nhiều cải tiến và thay đổi lớn. Do đó, chúng tôi không đủ thời gian để có thể đánh giá tương quan giữa Z3-regex và Z3-str2, đồng thời không thể làm lại các thí nghiệm được nhắc đến ở phần sau của khóa luận.

Chúng tôi ra mắt Z3-regex lần đầu tiên tại Hội nghị sinh viên nghiên cứu khoa học cấp Khoa, Khoa Công nghệ Thông Tin, Đại học Công nghệ, Đại học Quốc gia Hà Nội [2] là một phiên bản sơ khai, còn nhiều lỗi và chưa đánh giá cao phần thực nghiêm. Cho đến thời điểm hiện tại, tôi đã nâng cấp về mặt cài đặt của Z3-regex, đặc biệt là ở hàm $Star()$ giúp công cụ có thể giải được nhiều trường hợp hơn. Khi tôi đánh giá Z3-regex bằng cách so sánh với bộ giải S3, chúng tôi có thể giải được nhiều trường hợp mà S3 không thể giải hoặc giải sai.

Từ khóa: *Máy giải SMT, Ràng buộc xâu, Biểu thức chính quy.*

[4] `https://github.com/anhtrung93/Z3-str-Regex`
[5] `https://sites.google.com/site/z3strsolver/`
[6] `http://www.comp.nus.edu.sg/~trinhmt/S3/`

Table of Contents

List of Tables . . . . **ix**

List of Figures . . . . **ix**

ABBREVIATION . . . . **x**

1 INTRODUCTION **1**

1.1 Motivation 1

1.2 Contributions and thesis overview 3

1.2.1 Contribution 3

1.2.2 Thesis structure 3

2 PRELIMINARIES **5**

2.1 Symbolic string execution 5

2.1.1 Overview 5

2.1.2 String solvers 7

2.2 Z3 SMT solver 8

2.3 Z3-str solver 9

2.3.1 Overview 9

2.3.2 Example of string solving in Z3-str 9

2.4 S3 solver 11

2.5 Other related works 13

3 APPROACH 15

3.1 Specification of Z3-regex 15

3.1.1 Project goals 15

3.1.2 Overall design 16

3.1.3 Sequence diagram 17

3.1.4 System requirements 18

3.2 An Example 18

3.3 Constraint Language 19

3.4 Pre-processing 22

3.5 Search Processing 23

3.6 Dependence Analysis 24

4 EXPERIMENT AND EVALUATION 27

4.1 Experiment 27

4.1.1 Requisite 27

4.1.2 Installation guide 27

4.1.3 Experimental benchmarks 29

4.1.4 Performace Results 29

4.2 Evaluation 31

5 CONCLUSION AND FUTURE WORKS 35

5.1 Conclusion 35

5.2 Future works 35

List of Figures

1 Symbolic execution example 6

2 Architecture of Z3[3] 8

3 Example of String solving [1] 10

4 The design of Z3-regex 16

5 The sequence diagram of Z3-regex 17

6 Solving Code 1 in steps 18

List of Tables

1 Explanation of Figure 3 in steps 11

2 The grammar of input constraint language 20

3 The grammar of input constraint language 21

4 The grammar of input constraint language 22

5 Regular expression reduction rules 24

6 Overall results in comparison 29

7 Details comparison performance of S3 and Z3-regex 30

ABBREVIATION

S3 : Symbolic String Solver

AST : Abstract Syntax Tree

SMT : Satisfiability Modulo Theories

SAT : satisfiable

UNSAT : unsatisfiable

regex : regular expression

int : Integer

bool : Boolean

Chapter 1

INTRODUCTION

1.1 Motivation

In the last decade, web application or web app has become very popular along with the significant growth of the Internet. End-users enjoy using a web browser as a convenient client because the web app runs on it is compatible across platforms and able to update without either a distribution or more installations in the client-side. Nowadays, the most visited web applications include web-mail, social networks, online retail sales, wikis and so on. Web application provides a ton of services for web users everyday and sometimes it also processes sensitive data[1]. Those protected information should be safeguarded against unwarranted disclosure. However, it is also the reason why malicious users try to attack web app frequently.

According to the list of top 10 most critical web application security risks of OWASP 2013[2] , the two vulnerabilities are Injection flaws and XSS (Cross Site Scripting) flaws are placed on the first and the third, respectively. These flaws are caused primarily by the inappropriate using of input strings. Recently, a various string solvers are built to cover these problems. The importance of string solving is explained in [4].

Generally, string solving plays a massive role in security verification and bug detection. It implies a significant interest of several research groups about modeling and reasoning about strings. Therefore, a number of powerful string solvers have been introduced, including, e.g., HAMPI [5], Kaluza [6], CVC4 [7], Z3-str [1], Z3-str2 [8] and S3 [4]. These apply different approaches to solve satisfiablity problems over string equations and some ([4] [7] [8]) also support regular expression (RE) membership predicates. By the time we do this research, S3 is the most powerful string solver.

[1]`http://help.unc.edu/help/what-is-sensitive-data/`
[2]`https://www.owasp.org/index.php/OWASP_Top_Ten`

Therefore, in this thesis, the performance of Z3-regex is evaluated mostly in comparison with S3.

When we examined carefully S3[3], we found some points that can be improved so we develop our solver called Z3-regex. First, the S3 team mentioned their $Term : regexpr$ presenting a regular expression in [4] and several equations around it. However, their $Term : regexpr$ was not a Z3 sort[4]. Z3-str, CVC4, and S3 solver treats string natively as its primitive type, i.e. without abstractions or representation conversions, which was named as the native solvers in [8]. In these solvers particularly in S3, regular expression inputs has to be constructed from string constrains using concatenation $(.)$, union $(+)$ and Kleene star $(*)$. In other hand, regular expressions can be presented in string format (RE). This not only no longer required a convert step before hand, but also keep regex in the original form, which is familiar to the programmers.

Second, S3 theoretically can solve the equal membership between the primitive types that is string and a regular expression function that is $Star()$. This is mentioned in [4] as Z3-str-star, a component of S3 and an extension of Z3-str. However, since Z3-str has supported many high-level string operations, which appear frequently in web applications, it is necessary to support directly the equalization between a high-level string operation and a regular expression function.

Third, the S3 developers mentioned in their paper [4] that they have to modify Z3 API to get interaction between String theory and Arithmetic theory. These newly added API methods allow them to query the length of a string variable and relationship between the lengths of different string variables. Noted that S3 is built as a theory plugin of Z3 Prover[5]. Although it is open source, it can change its API in the future. Therefore, we suppose the modification to the core Z3 is not a sustainable development.

Fourth, there is not any benchmark for regular expression solving evaluation. As mentioned before, S3 has introduced and solved some regular expression functions, but it is not tested in cases. So it is necessary to provide a set of benchmarks for regular

[3]`http://www.comp.nus.edu.sg/~trinhmt/S3/`
[4]`http://research.microsoft.com/en-us/um/redmond/projects/z3/ml/z3.html`
[5]`https://github.com/Z3Prover/z3`

expression solving evaluation.

Hence, we address the above problems in our solver called Z3-regex. Headforemost, we need to construct a string solver that can integrate with Z3 SMT solver and solve regular expression constraints. We declare the input constraints as the mixture of regex and other primitive types such as bool, int, string. Next, we determine if the input constraints are satisfiable (SAT) or not (UNSAT). If SAT, we return a model - values of input variables that make the constraints satisfiable.

1.2 Contributions and thesis overview

1.2.1 Contribution

Overall, we develop our tool, Z3-regex on top of Z3-str that can solve regular expression constraints and provide new contraint language for regular expression in addition to existing rich syntax supporting string and non-string terms of Z3-str. We also register a standalone Z3 sort of regular expression in Z3-regex. We call it $regex$ and its simple form $simpleRegex$. An almost complete regex constrains parser is provided as well. In Z3-regex, we have solved frequently operations in regular expression using that are $Matches()$ and $Star()$ and the equalization between them and high-level string expression such as $Concat()$. Next, we created a small set of benchmarks for regular expression evaluation. Finally, Z3-regex[6] is publicly available for further dicussion and development.

1.2.2 Thesis structure

The remainder of this thesis is divided into 5 main parts organized as followed.

Chapter 2 provides background knowledge that related to string symbolic execution and string solvers base on automaton and SMT. The main points in this chapter are Z3 SMT solver's architecture and some string solver's approaches.

Then the chapter 3 mentions about my approach in Z3-regex and the implementation in details.

[6] `https://github.com/anhtrung93/Z3-str-Regex`

Chapter 4 will clearly specify how we did our experiments and the results with evaluation in comparison with S3 solver as well. We also instruct in details how to install and execute Z3-regex in this chapter.

The thesis ends with the conclusion and future works in chapter 5. We will propose some approaches that can improve Z3-regex to the completeness.

Chapter 2

PRELIMINARIES

2.1 Symbolic string execution

2.1.1 Overview

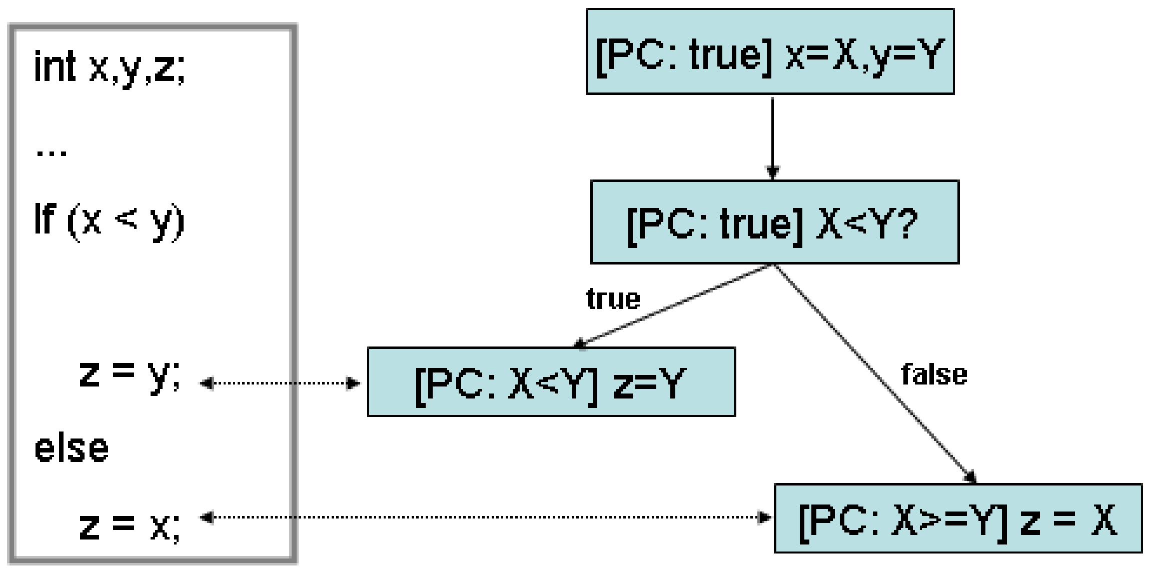
Symbolic execution is an analysis technique to determine if each path of a program is executed by what input. The concept of symbolic execution was introduced academically with descriptions of: the Select system [9], the EFFIGY system [10],the DISSECT system [11], and Clarke's system [12]. It is usually used for error finding in complex code or automated test generation. The most considerable point of symbolic execution is that it uses symbolic values instead of actual concrete data. For example, input variables are presented as symbol values, the output values are returned as a function of input symbolic values then use *path condition* to represent each class. A path condition can be considered as a conjunction of Boolean constraints without any quantifier over the symbolic inputs. Each bool formula stores a branch of execution tree that input variables must satisfy to reach a decision point. Symbolic execution tree constructs path conditions during the dynamic execution of the program. When the execution starts, there is only one path condition(PC) with true value:

$$PC(0) = true.$$

Then the execution observed the program code until it found the first branch condition q_0, the path condition is split into two. The first is appended with constraint q_0 and $\neg q_0$ with the second one.

$$PC(0.0) = true \wedge q_0$$
$$PC(0.1) = true \wedge \neg q_0$$

The path condition $PC(0.0)$ is considered as the true branch while $PC(0.1)$ is the false branch. The process continues till it reaches the end of a branch, it then follows other



In the first block on the right hand side, Path Condition (PC) is initially true with the value of x and y are X and Y, respectively. The second block is an if-then-else statement. If the program takes the true branch, the path condition will be $X < Y$ and z is assigned the value of Y in the third block. If the program takes the false branch, the path condition will be $X \geq Y$ and z now equal to X.

Figure 1: Symbolic execution example[1]

branches in depth-first model. After all path conditions has been constructed, each branch needs to be solved to get some values that represent to that branch. This in order to reproduce the program's execution.

The original use of symbolic execution applied to integers, boolean and even reference in object-oriented programs. After that, due to the popularity of web application, which do a lot of string processing, and the increasing risk that errors in web servers may lead to security violations, researchers started to develop tools that extend symbolic execution in string domain. Many techniques have been applied in recent solvers to solve string constrains that are mainly can be divided into two groups: SMT-based and automaton-based. The two approaches are investigated and evaluated in [13], will be discussed in the Section 2.1.2.

[1]`http://javapathfinder.sourceforge.net/extensions/symbc/doc/`

2.1.2 String solvers

A string solver is a tool that is used to solve string-related constraints. Even being a reference type, it uses string no less than other primitive types, which makes its solver as much as important as numeric solvers. As an important part of automated software testing and analysis of database/web applications, string solvers have received increasing interest recently.

2.1.2.1 Automaton-based method

Naturally, string and string constraints can be presented by automaton because string can be some words over some alphabet and string variable can store languages of words. Moreover, this approach determines a natural mapping from string operation to automaton operations. Many researches have proposed several methods in solving automaton equations translated from a given set of string constraints [13]. This approach is also very useful in solving regular expression constraints because finite automatons are often used to represent the regular languages as well. However, also from [13], the final automaton may not satisfy the first constrains if these string constraints are solved sequentially. Some SAT solvers using this approach are introduced in the Section 2.5.

2.1.2.2 SMT-based method

Satisfiability modulo theories(SMT) problem is a decision problem which is expressed in first-order logic. An SMT instance is presented in a form of a Boolean satisfiable instance where sets of variables represent predicates from underlying theories. An SMT solver can reason about list, arrays, bit vectors and so on. Generally, there are several third-party constraints solvers such as SAT solver, integer constraint solver and string constraint solver, that we consider, are built with the SMT solver layer on top. In this method, constraints can be expressed in several ways, implies the different approaches in a number of SAT solvers. We will consider some of these in the related works. Thereafter, encoded constraints are passed on to an SMT-based solver, which, if the problem is satisfiable, will return a map which represents one solution for any given variable.

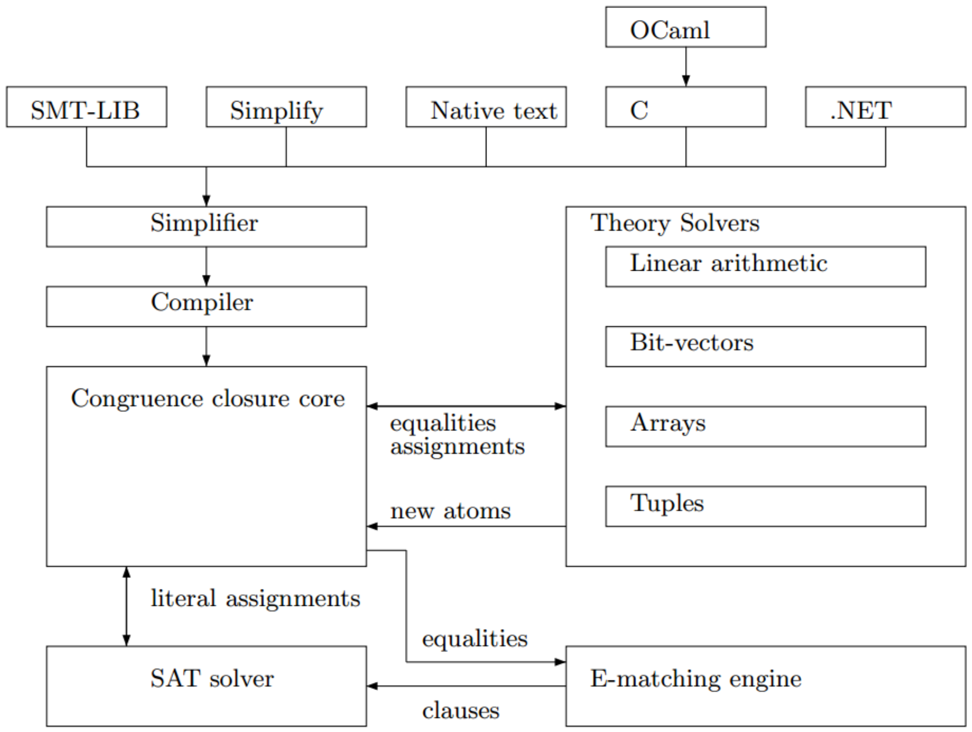


Figure 2: Architecture of Z3[3]

2.2 Z3 SMT solver

Among a dozen of available SMT solvers, Z3 SMT Solver [3] is currently one of the most powerful ones. As an SMT solvers, Z3 enable applications such as extended static checking, predicate abstraction, test case generation, and bounded model checking over infinite domains. Introduced by Microsoft Research, Z3 targets at solving problems in software verification and software analysis. There are two brief uses of Z3 that are Z3 allows end-users interact by using textual format, including SMT-LIB [14] and Simplify [15]. One can also call Z3 procedurally by using either an ANSI C API, an API for the .NET managed common language runtime, or an OCaml API.

Figure 2 describes generally the inner working of Z3, it integrates a modern DPLL-based SAT solver that handles the boolean structure of the input formula, a congruence closure core theory solver that handles equalities and uninterpreted functions, satellite solvers (for arithmetic, arrays, etc.), and an E-matching abstract machine (for quantifiers). The detailed explanation of each component can be found in [3]

2.3 Z3-str solver

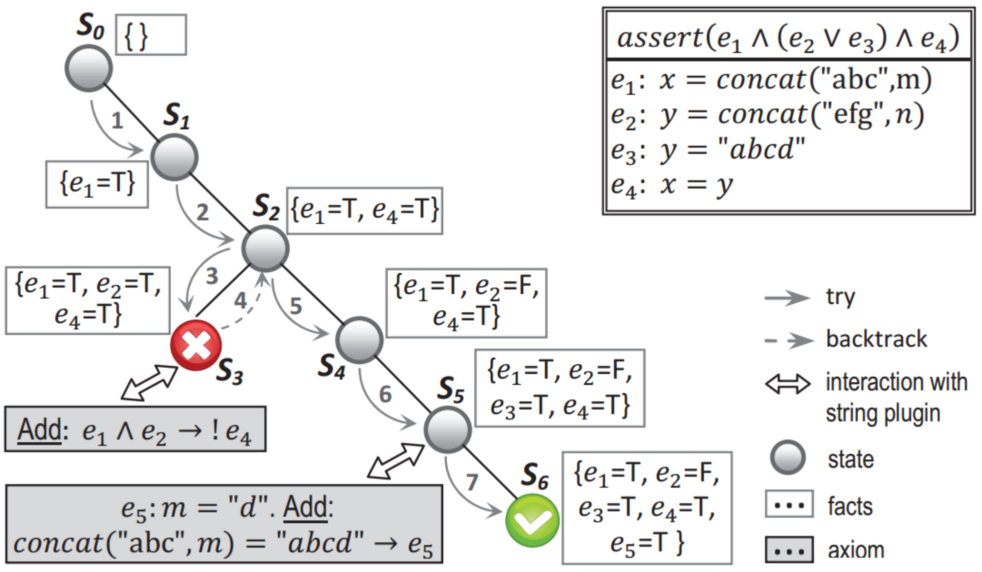
2.3.1 Overview

Z3-str [1] can be considered the first SMT-based string solver. Instead of relying on other theories, it builds a string theory for itself and allows this string theory to be plugged into a modern and powerful SMT solver Z3.

In [1], the authors present Z3-str as a satisfiability solver that supports a rich combined logic over strings and non-string operations aimed at symbolic, static and dynamic analysis of web applications. Z3-str treats strings as a primitive type, thus avoiding the inherent limitations observed in many other string solvers that encode strings in terms of other primitive such as determining string lengths before the solving process (fixed-bit vector), which can be very difficult through static analysis. More importantly, it allows Z3-str to support unbounded string variables and related operations that otherwise cannot always be supported if lengths have to be determined. The supported logic has three sorts, namely, bool, int and string. The string-sorted terms include string constants and variables of arbitrary length, with functions such as concatenation, sub-string, and replace. The int-sorted terms are standard, with the exception of the length function over string terms. The atomic formulas are equations over string terms and (in)-equalities over integer terms. Formulas are constructed in the usual way through Boolean combination of atomic formulas. Z3-str takes a formula in this logic as input, and decides if it is satisfiable.

2.3.2 Example of string solving in Z3-str

Next, we will explain the process of how Z3's core component and string theory solver Z3-str interact. We consider the string constraints on the table on the right in Figure 3. The core component cannot interpret the string operations; instead, it treats them as four independent Boolean variables (e_1, e_2, e_3, e_4) and tries to assign Boolean values to them. Table 1 lists out the steps in details for the example in Figure 3. There is no fact or axiom initially in state S_0. The core starts by setting e_1 and e_4 to true and reaches state S_2. Then, assume the core tries true for e_2 before assigning e_3 at state S_3. Recall that the core can detect functionally equivalent terms (i.e., based on the theory of uninterpreted functions).



S_0 {}
S_1
$\{e_1$=T$\}$
S_2 $\{e_1$=T, e_4=T$\}$
$\{e_1$=T, e_2=T, e_4=T$\}$
S_3
Add: $e_1 \wedge e_2 \rightarrow !e_4$
S_4 $\{e_1$=T, e_2=F, e_4=T$\}$
S_5 $\{e_1$=T, e_2=F, e_3=T, e_4=T$\}$
$e_5: m =$ "d". Add: $concat$("abc", m) = "abcd" $\rightarrow e_5$
S_6 $\{e_1$=T, e_2=F, e_3=T, e_4=T, e_5=T $\}$
1
2
3
4
5
6
7
$assert(e_1 \wedge (e_2 \vee e_3) \wedge e_4)$
e_1: $x = concat$("abc",m)
e_2: $y = concat$("efg",n)
e_3: $y =$ "abcd"
e_4: $x = y$
try
backtrack
interaction with string plugin
state
facts
axiom

Figure 3: Example of String solving [1]

Hence, from the facts $e_1 = true$, $e_2 = true$ and $e_4 = true$, the core puts x; y; "abc".m; and "efg".n into one equivalence class and notifies the plug-in. The plug-in thus knows the two concat (.) above are equivalent.

However, from the semantics of concatenation, these two concats cannot equal to each other under any circumstances because they do not share a same prefix. The plug-in informs the core about the new search through an axiom($e_1 \wedge e_2 \Rightarrow \neg e_4$). With the new axiom and the existing facts, the core detects a conflict on e_4 (in the bool domain). The core backtracks to state S_2 and tries the other option for e_2. Note that when the core backtracks, it discards the recent fact and any insertions into equivalence classes as the consequence of the fact. Then, the core assigns true to e_3 so that "$abcd$" will be put in the same equivalence class as "abc".m. Again, based on the concatenation semantics, the value of string variable m can be inferred by the string theory plug-in, which must be "d". This new finding is formulated by introducing a new variable e_5 in an axiom "abc".m ="$abcd$"$\Rightarrow e_5$, which is sent back to the core. The new state is S_6 in the figure. From the existing facts and the new axiom, the core derives e_5 is true.

	Fact added	Equivalent classes	Reduction/Action
1	$x = concat(\text{“}abc\text{”}, m)$	$\{ x; \text{“}abc\text{”}.m \}$	
2	$x = y$	$\{ x; y; \text{“}abc\text{”}.m \}$	
3	$y = concat(\text{“}efg\text{”}, n)$	$\{ x; y; \text{“}abc\text{”}.m;$ $\text{“}efg\text{”}.n \}$	- Conflict detected - Backtrack and remove facts - Try another option for e_2
4	$y = \text{“}abcd\text{”}$	$\{ x; y; \text{“}abc\text{”}.m;$ $\text{“}efg\text{”}.n; \text{“}abcd\text{”} \}$	$\text{“}abc\text{”}.m = \text{“}abcd\text{”} \Rightarrow m = \text{“}d\text{”}$
SAT solution: $x = \text{“}abcd\text{”}, y = \text{“}abcd\text{”}, m = \text{“}d\text{”}$			

Table 1: Explanation of Figure 3 in steps

At state S_6, all Boolean expressions have been assigned and the assignments are consistent. Besides, the satisfying values of string variables x, y and m can be retrieved from their equivalence classes. Therefore, a set of consistent and satisfying solutions for the input constraint has been found and the search procedure terminates.

2.4 S3 solver

S3 [4] stands for Symbolic String Solver, is published in 2014. According to the papers, there is a real requirement for reasoning about unbounded strings. In verifying client-side input validation functions, a bounded string solver (such as Kaluza [6] and Z3-str [1]) can only find policy violations but it cannot prove the conformance to a given policy. There are certainly some solvers [16] [17] [18] [19] that can reason about un-bounded strings. However, their key weakness is that they cannot handle non-string constraints, particularly length constraints. The research also states some statistics from a comprehensive study of practical JavaScript application [6]. Constraints arising from the applications have an average (per benchmark query) of 63 JavaScript string operations, while the remaining are boolean, logical and arithmetic constraints. The largest fraction is for operations like $indexOf$, $length$ (78%). A significant fraction of the operations, including $substring$ (5%), $replace$ (8%), and $split$, $match$ (1%). In the $match, split$ and $replace$ operations, 31% are based on regular expressions. As shown in the statistics above, missing length constraints (whose appearance is frequent) will lead to many false positives. This clearly is not acceptable.

To solve all problems above, the researchers conclude S3 with three important features. First, S3 can handle unbounded regular expression in the presence of length constraints, and express precisely high-level string operations, which ultimately enables a more accurate string analysis. There are three primitive types in S3: integer, boolean and string. The input formula can be of the following forms:

- A boolean expression
- A comparison operation between two integers or boolean expressions
  An equation between two string expressions. S3 also supports other common string operations.
- An equation between two string expressions. S3 also supports other common string operations.
- A membership predicate between a string expression and a regular expression, where an expression can either be a string constant, a variable or their concatenation, and regular expressions are constructed from string constants using $concatenation(.)$, $union(+)$ and $Kleene$ star $(*)$.
- A composite formula constructed using negation and binary connectives, including $\wedge, \vee, \Rightarrow$

Second, S3 can detect more vulnerabilities and bugs, in comparison with Kaluza – the core of Kudzu and Z3-str, which are important existing solvers that can support both string and non-string operations, especially the length constraint. Compared to the constraint syntax of Z3-str, S3 can be viewed as an extension with regular expressions, membership predicates, and high-level string operations that often work on regular expressions such as $search, replaceAll, match, split, test, exec$. S3's constraint language is also slightly more expressive than Kaluza's since it handles above string operations in its original semantics – unbounded.

Third, S3 provides an algorithm for string theory is designed in fashion driven by the try-and-backtrack procedure of the Z3 core. So that given a set of input constraints, it performs incremental reduction for string variables until the variables are bounded with constant strings/characters. Noted that, the design S3 is inspired most by Z3-str, so S3 is Z3-based solver as well.

2.5 Other related works

HAMPI [5] is one of the most popular open-source string solvers. It was originally designed for detection of SQL injection vulnerabilities. It was then widely used in other web application analysis [6]. HAMPI works only with string constraints over fixed-size string variables. It extends the constraint language to membership in fixed-size context-free languages, but considers only problems over one string variable. Input problems are reduced first to bit-vector problems and then to SMT Solver.

Kaluza is the core of a JavaScript dynamic test generation framework Kudzu [6]. It extends both STP[2] and HAMPI, and supports int, boolean and string constraints generated from an execution path. Kaluza leverages HAMPI's front-end to model strings as bit-vectors so they can be reasoned uniformly with other types of constraints in STP. However, a prerequisite is that the lengths of bit-vectors have to be known beforehand. Hence, before solving the string values, Kaluza first tries to find a satisfying solution to string lengths. Then it encodes string constraints based on the concrete length values. However, string lengths vary from path to path, and sometimes may be unknown in the static analysis context.

Automata/regular-expressions are a natural form to represent strings so that many works are based on them. Java String Analyzer[3] (JSA) applies static analysis to model flow graphs of Java programs [20]. These graphs capture dependencies between string variables. Finite automata can be computed from the graphs to reflect possible string values. The work in [21] used finite state machines (FSMs) for abstracting strings during symbolic execution of Java programs. They handle a few core methods in the

[2]http://stp.github.io/
[3]http://www.brics.dk/JSA/

java.lang.String class, and some other related classes. They partially integrate a numeric constraint solver. For instance, string operations which return integers, such as *indexOf*, trigger case-splits over all possible return values. This approach is very useful in solving regular expression constraints because finite automatons are often used to represent the regular languages and a regular expression and a regular language are equivalent. However, most of existing approaches have difficulties in handling string operations related to integers such as *length* and *indexOf*.

While bit-vectors and automaton-based methods have been developed for a long time, parameterized-array based method has firstly been considered in [22]. The basic idea of this approach is using a parameterized array to model strings. PASS standing for Parameterized Array based String Solver converts all string constraints into quantified expressions. In the case that the string has no initial form of regular expression, it is converted directly to P-array constraint. On the other hand, when a regular expression constraint exists, the string will be first converted into automata in order to do some refinement steps, and then again, it will be transformed into quantified expressions. Recently, CVC4 [7] has presented an alternative approach, based on algebraic techniques for solving (quantifier-free) constraints natively over a theory of unbounded strings with length and regular language membership. CVC4 also integrate into general, multi-theory SMT solvers, which based on DPLL(T) architecture [23]. The researchers also introduce CVC4 is the first solver able to reason about a language of mixed constraints that includes strings together with integers, reals, arrays, and algebraic data types as well.

Chapter 3

APPROACH

In this chapter, I will firstly state the problems in the specification of Z3-regex. The overall design of the tool will be introduced in this section as well. After that, I present a simple example to make my work clearer. The last three sections will describe my approach in detailed steps.

3.1 Specification of Z3-regex

3.1.1 Project goals

There are some goals I aim to reach in this project. First, I extend Z3-str [1] to Z3-regex with the additional regular expression solving. To be more specified, Z3-regex needs to support regular expression in the input constraints and be able to solve some frequently used functions of regular expression. Because the existing representation of S3 [4], as mentioned in the previous section 1.1, requires a step beforehand and seems to be less familiar to programmers, it is needed to represent regular expression in a more friendly way.

Second, since the solved equal membership between a regular expression and other string functions is still insufficient in existed string solvers, I declare some regex functions and propose a theoretical approach to solve these functions and their equalization with some main high-level string functions.

Third, it is necessary to create a benchmarks set for regular expression solver testing. The most common benchmark suite is Kaluza Benchmark suite. The Kaluza constraints were generated by a JavaScript symbolic execution engine [6], where length, concatenation and (finite) RE membership queries occur frequently. To my knowledge, this is the only publicly available string constraints set. Therefore, there should be at least a dozen of test cases to compare Z3-regex with S3 solver in regex solving.

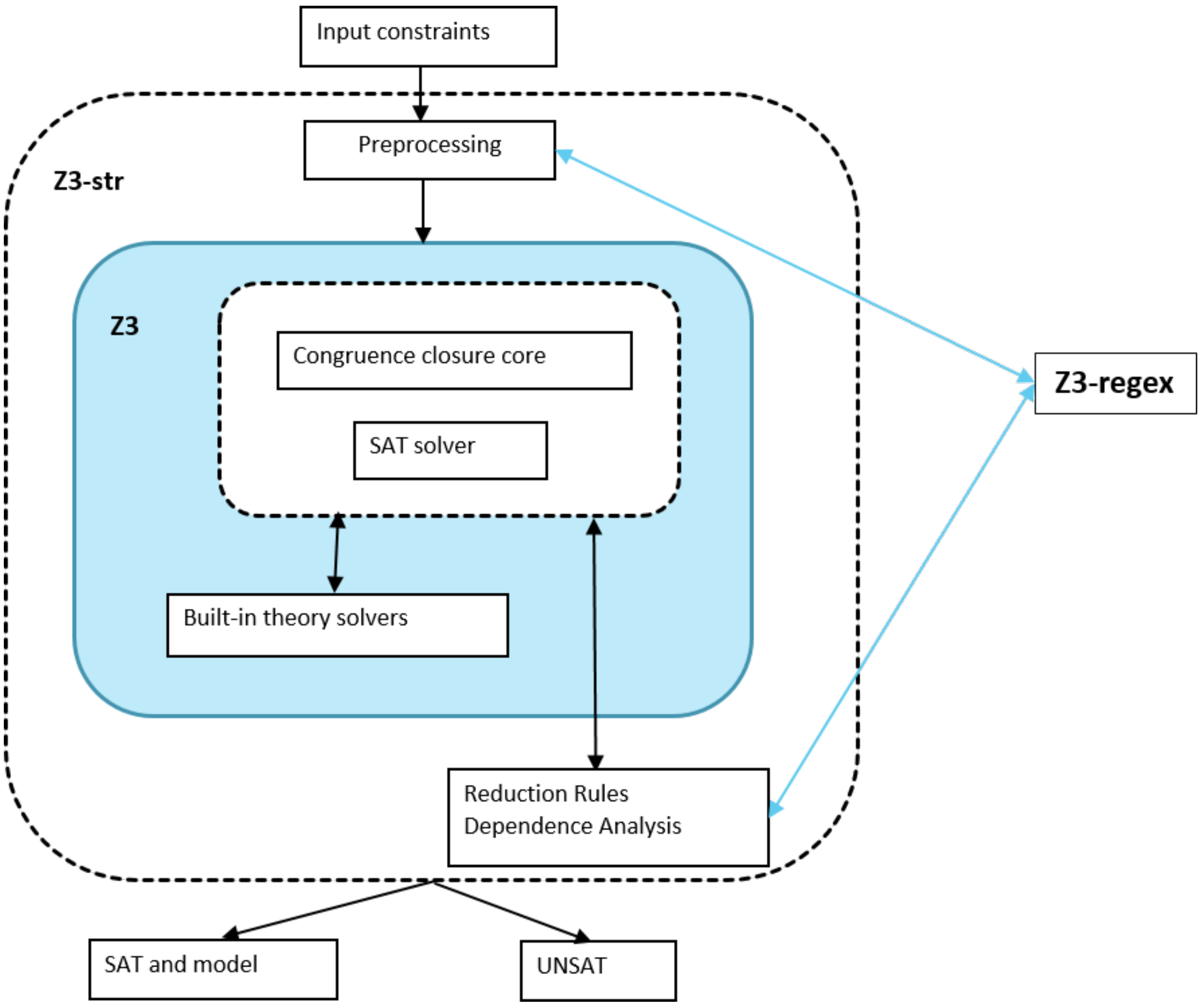


Figure 4: The design of Z3-regex

3.1.2 Overall design

Because Z3-regex is extended from Z3-str, it acts as a plug-in string theory for an SMT solver Z3 [3] and communicates with the Z3-core via Z3 API. Overall, there are three main parts of Z3-regex as illustrates in Figure 4: Pre-processing, Search processing and Dependence analysis. At first, Z3-regex converts string function into a group of main functions. Next, we apply our reduction rules, support the searching process of Z3 core and reduce formulas into simpler equations. Finally, the plug-in theory analyzes the dependence graph from the current context and assign free string and integer variables. All above parts will be discussed in section 3.4, 3.5 and 3.6.

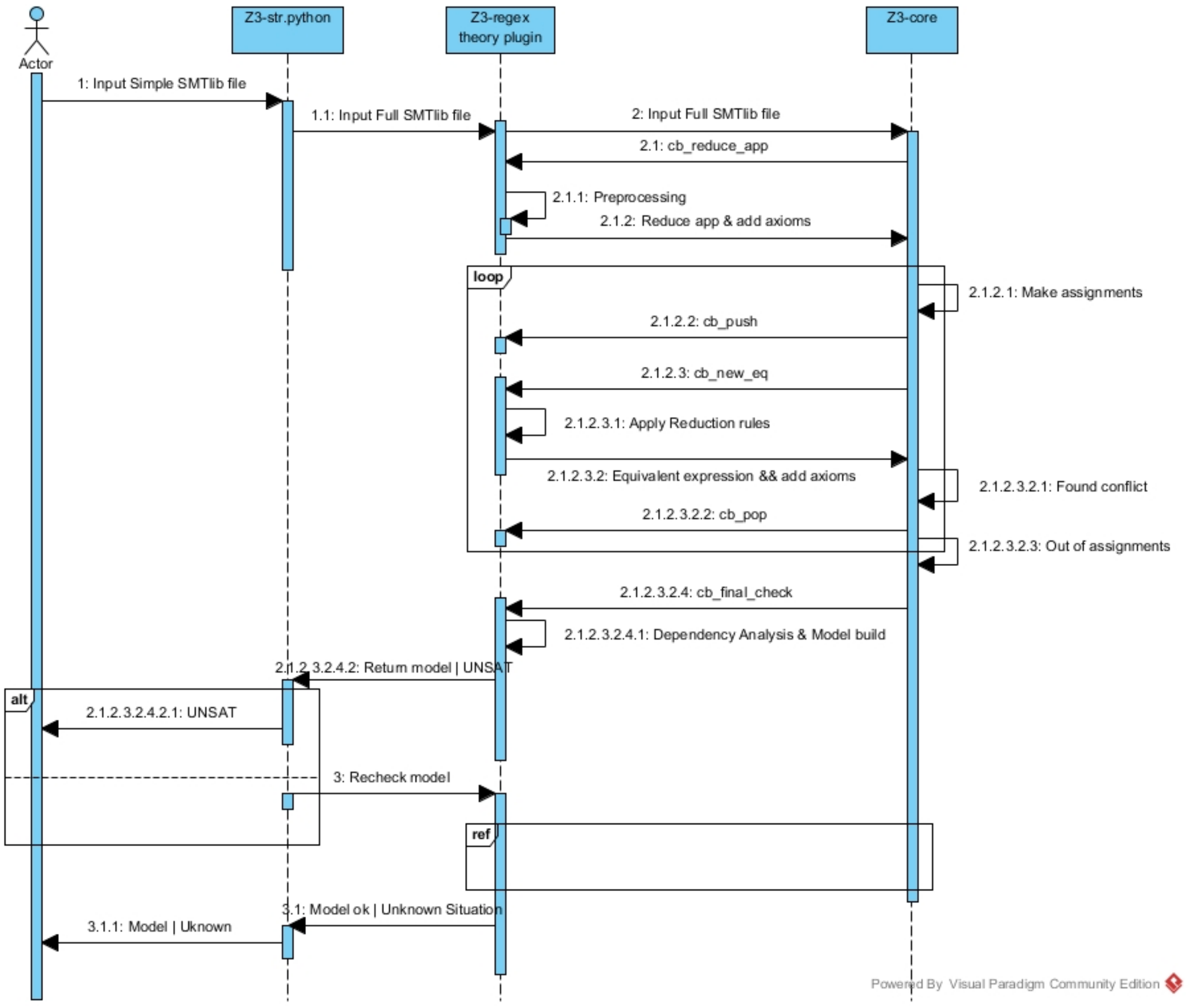

Actor
Z3-str.python
Z3-regex theory plugin
Z3-core
1: Input Simple SMTlib file
1.1: Input Full SMTlib file
2: Input Full SMTlib file
2.1: cb_reduce_app
2.1.1: Preprocessing
2.1.2: Reduce app & add axioms
loop
2.1.2.1: Make assignments
2.1.2.2: cb_push
2.1.2.3: cb_new_eq
2.1.2.3.1: Apply Reduction rules
2.1.2.3.2: Equivalent expression && add axioms
2.1.2.3.2.1: Found conflict
2.1.2.3.2.2: cb_pop
2.1.2.3.2.3: Out of assignments
2.1.2.3.2.4: cb_final_check
2.1.2.3.2.4.1: Dependency Analysis & Model build
2.1.2.3.2.4.2: Return model | UNSAT
alt
2.1.2.3.2.4.2.1: UNSAT
3: Recheck model
ref
3.1: Model ok | Unknown Situation
3.1.1: Model | Uknown
Powered By Visual Paradigm Community Edition

Figure 5: The sequence diagram of Z3-regex

3.1.3 Sequence diagram

The sequence diagram models the collaboration of objects based on a time sequence. It shows how the objects interact with others in a particular scenario of a use case. Created by Visual Paradigm, Figure 5 illustrates the detailed the possible processes from the inputs to the outputs in Z3-regex.

Step	Fact	Eq-class	Reduction/Action/New formula	In process
1	$x \in$ abc[def]*	$\{x, \text{"abc"}.\mathit{star}([\text{def}], n)\}$	$\rightarrow x = \text{"abc"}.\mathit{star}([\text{def}], n) \wedge n \geq 0$	Preprocess
2	$\mathit{EndsWith}(x, \text{"}f\text{"})$	$\{x, \text{"abc"}.$ $\mathit{star}([\text{def}], \text{n}), s_1.\text{"f"}\}$	$\rightarrow x = s_1.\text{"f"} \wedge \mathit{length}(s_1) \geq 0$	
3		$\{x, \text{"abc"}.$ $\mathit{star}([\text{def}], \text{n}), s_1.\text{"f"}\}$	$\text{"abc"}.\mathit{star}([\text{def}], \text{n}) = s_1.\text{"f"}$ $\rightarrow (s_1 = \text{"abc"}.s_2 \wedge \mathit{star}([\text{def}],\text{n}) = s_2.\text{"f"} \wedge \text{length}(s_1) \geq 3)$ $\vee (s_1 = \text{"abc"} \wedge \mathit{star}([\text{def}],\text{n}) = \text{"f"})$	Search process
4		$\{x, \text{"abc"}.$ $\mathit{star}([\text{def}], \text{n}), s_1.\text{"f"}\}$ $\{s_1, \text{"abc"}.s_2\}$	Identify simple equations: $x = \text{"abc"}.\mathit{star}([\text{def}], n)$ $x = s_1.\text{"f"}$ $s_1 = \text{"abc"}.s_2$ $\mathit{star}([\text{def}],\text{n}) = s_2.\text{"f"}$	Dependence analysis
5			Try to assign $n = 0 \rightarrow x = \text{"abc"}$, conflicts with $x = s_1.\text{"f"}$ Try to assign $n = 1 \rightarrow x = \text{"abcd"} \vee x = \text{"abce"} \vee x = \text{"abcf"}$ Choose to assign $x = \text{"abcf"} \rightarrow s_1 = \text{"abc"} \rightarrow s_2 = \text{"f"}$	
SAT solution: $x = \text{"abcf"}$				

Figure 6: Solving Code 1 in steps

3.1.4 System requirements

At the moment, Z3-regex is confirmed to work on Ubuntu 14.04[1]. It was implemented in C++ and integrated with Z3 SMT via Z3 API [2].

3.2 An Example

We illustrate our approach via a simple example input constraint in Code 1. Here `x` is a string variable that should match the regular pattern `'abc[def]*'` and end with the string `"f"`. We can check that `abcdef` is a model for the constraint. Simplified solving steps are shown in Figure 6. We will explain these steps in more detailed in the following sections.

Code 1: A sample constraints

```
(declare-variable x String)
(assert (= true (Matches x 'abc[def]*') ) )
(assert (= true (EndsWith x "f") ) )
```

[1]`http://releases.ubuntu.com/14.04/`
[2]`http://research.microsoft.com/en-us/um/redmond/projects/z3/ml/z3.html`

3.3 Constraint Language

In comparison with Z3-str, Z3-regex supports regular expression and two functions that appear frequently that are $Matches()$ and $Star()$. Therefore, I expand the input language in Z3-regex, so it can accept regex constraints. As mentioned previously, S3 solver only support regular expression in the form of Concatenation $(.)$, Union $(+)$ and Kleene star $(*)$ functions as input. This approach requires an additional step to convert regular expressions of its original string form, used by web applications, to S3's function format.

Hence, I introduce the constraint language of Z3-regex in Table 2. For simplicity, we only list primitive types: int, bool, string and $regex$ in addition. As can be seen in the Table 2, Z3-regex provide new boolean function $matches()$ and a new function $star()$ equal to a string. Since I have introduced a new sort for regular expression called regex, $Term : regex$ is declared to equal to $< RE >$. $< RE >$ is the regular expression in string format, it is also listed in advanced in the Table 3.

The input formula of Z3-regex can be one of the following forms:

- Boolean expression
- A comparison operation between two integers or boolean expressions an equation between two string expressions
- A composite formula constructed using negation and binary connectives.
- A membership predicate between a high-level string operations and a regular expression function like $Matches()$ and $Star()$.

⟨*Term:bool*⟩	::=	⟨*Var:bool*⟩
	|	TRUE
	|	FALSE
	|	**contains**(⟨*Term:string*⟩,⟨*Term:string*⟩)
	|	**matches**(⟨*Term:string*⟩,⟨*Term:regex*⟩)
⟨*Term:int*⟩	::=	⟨*Var:int*⟩
	|	⟨*number*⟩
	|	⟨*Term:int*⟩ {+, -, *, /} ⟨*Term:int*⟩
	|	**length**(Term:string>)
	|	**indexof**(⟨*Term:string*⟩,⟨*Term:string*⟩)
⟨*Term:regex*⟩	::=	⟨*RE*⟩
⟨*Term:string*⟩	::=	⟨*Var:string*⟩
	|	⟨*constString*⟩
	|	**concat**(⟨*Term:string*⟩,⟨*Term:string*⟩)
	|	**substring**(⟨*Term:string*⟩,⟨*Term:int*⟩,⟨*Term:int*⟩)
	|	**replace**(⟨*Term:string*⟩,⟨*Term:string*⟩,⟨*Term:string*⟩)
	|	**star**(⟨*Term:regex*⟩,⟨*Term:int*⟩)
⟨*Expr:bool*⟩	::=	⟨*Term:bool*⟩
	|	⟨*Term:int*⟩ {⟨, ⟩, $\leq$, $\geq$} ⟨*Term:int*⟩
	|	⟨*Term:string*⟩ = ⟨*Term:string*⟩
	|	not ⟨*Expr:bool*⟩
	|	⟨*Expr:bool*⟩ $\wedge$ ⟨*Expr:bool*⟩
	|	⟨*Expr:bool*⟩ $\vee$ ⟨*Expr:bool*⟩
	|	**if** ⟨*Expr:bool*⟩ **then** ⟨*Expr:bool*⟩ **else** ⟨*Expr:bool*⟩
	|	⟨*Expr:bool*⟩ $\Rightarrow$ ⟨*Expr:bool*⟩
⟨*Assertion*⟩	::=	assert ⟨*Expr:bool*⟩

Table 2: The grammar of input constraint language

```
⟨RE⟩                ::=  ⟨union⟩
                      |  ⟨simple-RE⟩

⟨union⟩             ::=  ⟨RE⟩⟨RE⟩

⟨simple-RE⟩         ::=  ⟨concatenation⟩
                      |  ⟨basic-RE⟩

⟨concatenation⟩     ::=  ⟨simple-RE⟩⟨basic-RE⟩

⟨basic-RE⟩          ::=  ⟨star⟩
                      |  ⟨plus⟩
                      |  ⟨question⟩
                      |  ⟨counter⟩
                      |  ⟨elementary-RE⟩

⟨elementary-RE⟩     ::=  ⟨group⟩
                      |  ⟨any⟩
                      |  ⟨char⟩
                      |  ⟨set⟩

⟨star⟩              ::=  ⟨elementary-RE⟩*

⟨plus⟩              ::=  ⟨elementary-RE⟩+

⟨counter⟩           ::=  ⟨elementary-RE⟩{⟨number⟩}
                      |  ⟨elementary-RE⟩{⟨number⟩,}
                      |  ⟨elementary-RE⟩{⟨number⟩,⟨number⟩}

⟨question⟩          ::=  ⟨elementary-RE⟩?

⟨group⟩             ::=  (⟨RE⟩)
```

Table 3: The grammar of input constraint language

3.4 Pre-processing

In this step, Z3-regex reduces constraints and equations to simpler equivalence. This is in order to improve the following processes. To simplify constraints, I apply some reduction rules and transform all the equations to a set of core functions that are $length()$ $concat()$ and $star()$. Noted that, Z3-str also has some preprocessing rules to convert high-level string operations into set of $length()$ and $concat()$ equations [1].

Since Z3-regex accepts a regular expression in string format, I also integrate a parser which applies some regular expression rules to convert a $< RE >$ to a set of string constraints satisfied by every string matching the $< RE >$. The reduction rules are shown in Table 4.

	Condition	Action
1	$s \in< char >$	$\Rightarrow s =< char >$
2	$s \in [< set-item >]$	$\Rightarrow \textbf{length}(s) = 1 \wedge (\forall c \in< set-item >, s = c)$
3	$s \in< RE_1 >\|< RE_2 >$	$\Rightarrow s \in< RE_1 >\vee s \in< RE_2 >$
4	$s \in< simple-RE >$ $< basic-RE >$	$\Rightarrow s_1 \in< simple-RE >\wedge s_2 \in< basic-RE >\wedge$ $\textbf{concat}(s_1, s_2) = s$
5	$s \in< elementary-RE > *$	$\Rightarrow s = \textbf{star}(< elementary-RE >, n) \wedge n \geq 0$
6	$s \in< elementary-RE > +$	$\Rightarrow s = \textbf{star}(< elementary-RE >, n) \wedge n \geq 1$
7	$s \in< elementary-RE >?$	$\Rightarrow s =$“” $\vee s \in< RE >$
8	$s \in< elementary-RE >$ $\{< number_1 >, < number_2 >\}$	$\Rightarrow s = \textbf{star}(< elementary-RE >, n) \wedge n \geq$ $number_1 \wedge n \geq number_2$
9	$s \in< elementary-RE >$ $< number >$	$\Rightarrow s = \textbf{star}(< elementary-RE >, n) \wedge n \geq$ $number$

Table 4: The grammar of input constraint language

Come back to our example, at Figure 6, step 1 and 2 are in pre-processing stage. At step 1, since variable x matches with ‘$abd[def]*$’, we have a new formula based on our reduction rules: $x =$“abc”.$\textbf{star}([def], n) \wedge n \geq 0$. Similarity, a new formula is generated in step 2: $x = s_1$.“f”$\wedge$ $length(s_1) \geq 0$. After these 2 steps, we also have the equivalent classes: $\{x,$“abc”$.star([def], n), s_1.$‘f’$\}$.

3.5 Search Processing

In this step, Z3-regex applies some reduction rules to string, integer and regular expression constraints to support the search process in the Z3 core. We provide a new bundle of rules for Star() functions because after pre-processing steps, we have all equations in form of $star()$, $length$ and $concat()$ mixture. The complete rules are shown in Table 5.

Moreover, we also divide $term : regex$ into two types that are simple regex and complex regex to enhance the solving performance. We define simple regex as a regular expression with only one possible string solution; a complex regex, on the other hand, has more than one satisfiable string. In other words, after applying our regex reduction rules in pre-processing stage, simple regex returns only one constraint string with a constant length. With a constant length, solving for the number of occurrences of that simple regex is much faster when dealing with constant strings or their concatenation functions. For example, $Term : regex$ 'abc' is a simple regex because there is only $Term : string$ "abc" matches with it.

Now we consider our example in Section 3.2 again with step 3. From the equivalent class, we have an equation that is: "abc".$star([def], n) = s_1$."f". It applies some reduction rules, then implies:

$(s_1 =$"abc".$s_2 \wedge star([def], n) = s_2$."$f$"$\wedge$ $length(s_1) \geq 3)$ (1)

or $(s_1 =$"abc"$\wedge$ $star([def], n) =$"f"$)$ (2)

From this new formula, the Z3 SMT treats it as two separated branches (1) and (2). Theoretically, Z3 will "push" into one branch, then do a backtrack if the core found any conflicts in it. Without losing generality, we assume Z3 tries branch (1) first. Therefore, beside the existed equivalent class in the previous step, now we have $\{s_1$,"abc".$s_2\}$ in another equivalent class as well.

$star(r_1, v_1) = x_1.s_1 \Rightarrow (star(r_1, v_1 - 1) = x_1.s_2 \wedge s_2.s_3 = s_1 \wedge s_3.matches(r_1) \wedge v_1 \geq 1) \vee (star(r_1, v_1 - 1).x_2 = x_1 \wedge (x_2.s_1).matches(r_1) \wedge v_1 \geq 1)$
$star(r_1, v_1) = s_1.x_1 \Rightarrow (star(r_1, v_1 - 1) = s_2.x_1 \wedge s_3.s_2 = s_1 \wedge s_3.matches(r_1) \wedge v_1 \geq 1) \vee (x_2.star(r_1, v_1 - 1) = x_1 \wedge (s_1.x_2).matches(r_1) \wedge v_1 \geq 1)$
$star(r_1, v_1) = star(r_2, v_2).x_1 \Rightarrow (v_1 = v_2 = 0 \wedge x_1 =$ “”$) \vee (v_2 = 0 \wedge x_1 = star(r_1, v_1)) \vee (v_1 \geq 1 \wedge v_2 \geq 1 \wedge x_2.star(r_1, v_1 - 1) = x_3.star(r_2, v_2 - 1).x_1 \wedge x_2.matches(r_1) \wedge x_3.matches(r_2))$
$star(r_1, v_1) = x_1.star(r_2, v_2) \Rightarrow (v_1 = v_2 = 0 \wedge x_1 =$ “”$) \vee (v_2 = 0 \wedge x_1 = star(r_1, v_1)) \vee (v_1 \geq 1 \wedge v_2 \geq 1 \wedge star(r_1, v_1 - 1).x_2 = x_1.star(r_2, v_2 - 1).x_3 \wedge x_2.matches(r_1) \wedge x_3.matches(r_2))$
$star(r_1, v_1) = x_1.x_2 \Rightarrow (x_1 =$ “”$\wedge x_2 =$ “”$\wedge v_1 = 0) \vee (x_1 = star(r_1, v_2) \wedge x_2 = star(r_1, v_1 - v_2) \wedge v_2 \geq 0 \wedge v_1 \geq v_2) \vee (x_1 = star(r_1, v_2).x_3 \wedge x_2 = x_4.star(r_1, v_1 - v_2 - 1) \wedge x_3.x_4.matches(r_1) \wedge v_2 \geq 0 \wedge v_1 - 1 \geq v_2)$
$star(sr_1, v_1) = star(sr_2, v_2) \Rightarrow v_1 * length(sr_1) = v_2 * length(sr_2)$
$star(sr, v_1) = star(r, v_2) \Rightarrow v_1 = v_2 = 0 \vee x_1.star(sr, v_1 - 1) = x_2.star(r, v_2 - 1) \wedge x_2.matches(r) \wedge x_1.matches(sr) \wedge v_1 \geq 1 \wedge v_2 \geq 1$

Table 5: Regular expression reduction rules

3.6 Dependence Analysis

In the above sections, we have introduced my reduction rules, which actually is the axiom addition to the core of Z3 with the form of $AST \rightarrow AST$ with AST and AST' are the original and the transformed syntax trees, respectively. Recall that Z3 does not understand string and regex types, so AST and AST' are treated as independent bool variables. For example, if we assign a true value to AST, the core also assigns a true value to AST' as well.

When no more reduction can be applied, the core reaches the form of $simpleequation$. If each equivalence class has a constant in one side, the solver will terminate with a SAT solution. Otherwise, it builds a dependence graph of variables involved in simple equations. Z3-str [1] provides equations in form of $x = Y$ and Y have no more than one non-singleton compound string terms. After that, they conclude x depends on all the variables inside Y.

From the dependence graph, they identify free variables, the variables that do not depend on others and do not have constants in their equivalence class. They then assign concrete values to free variables by adding new axioms.

However, a free variable x in the string domain may be constrained by length assertions (in the integer domain) on the variable itself or on other variables that are dependent on x. Since length constraints do not constrain string values but rather their length, Z3-str introduces a free variable rule $(\mathit{freeVar})$ to allow the core to try to assign predefined constant strings of various lengths to a free variable in order to satisfy length constraints. According to the rule, if the plug-in detects that a string variable x is a free variable in the current context, it adds an axiom that states that the current context implies that x may have constant strings of various lengths. The context is the conjunction of all the facts set by the core up to this point. The authors have to use the context as the antecedent as x may not be a free variable in a different context. Any conflicts by length constraints will cause the core to try a constant string of different length.

Since we built Z3-regex on top of Z3-str and S3 does not mention about this step, we inherit the dependence analysis from Z3-str and improve it to analyze our regular expression constraint and its operations as well. As the $Star()$ function cannot be reduced in previous steps, we now will example the dependence of its variables. Normally, $star()$ has an integer variable in this step and Z3-regex will assign various incremental concrete values for integer variables, then add the new $Star(term : regex, term : int)$ with a known integer as an axiom to Z3. If the assignment value reaches a specific value of 50, Z3-regex supposes the equations are unsatisfiable.

To have a clearer view at this stage, we will consider step 4 and 5 in Figure 6 from the example in Section 3.2. From previous steps, we can identify a bundle of simple equations:

1. $x =$"abc"$.star([def], n).n)$
2. $x = s_1.$"f"
3. $s_1 =$"abc"$.s_2$
4. $star([def], n) = s_2$"f"

Firstly, we try to assign $n = 0$, it implies that $star([def], n) = star([def], 0) =$“”. From (1.), we can find $x =$“abc”. However, this value conflicts with $x = s_1$.“f” in (2.). Therefore, Z3 do a backtrack and try an incremental value of n.

Secondly, we try to assign $n = 1$, it implies that $star([def], n) = star([def], 1)$. This new $star()$ is parsed by the parser into: $x =$“$abcd$”$\vee$ $x =$“$abce$”$\vee$ $x =$“$abcf$”. The Z3 core tries each branch, checks if there is any conflict. After "push" and "pop" several times, Z3 found $x =$“$abcf$” satisfying all 4 simple equations above. That value of x implies $s_1 =$“abc” and $s_2 =$“f”.

Finally, Z3-regex returns a SAT solution: $x =$“$abcf$”.

Chapter 4

EXPERIMENT AND EVALUATION

This chapter will mostly illustrate my experiment with Z3-regex and its comparison with S3 solver.

4.1 Experiment

4.1.1 Requisite

The experiment was taken under the requisite

- Ubuntu 14.04 [1]
- Z3 4.1.1 [2]
- Lastest Z3-regex [3]
- S3 [4]
- Boost C++ Libraries[5] (version 1.57.0)
- Intel®Core™i5-2430M CPU @ 2.4GHz x 4
- Memory: 3.8 GB

4.1.2 Installation guide

Firstly, we need to install Z3 SMT, Boost Libraries and Z3-regex before running the experiment. This subsection will guide how to build Z3 SMT Solver, Boost Libraries and

[1] `http://releases.ubuntu.com/14.04/`
[2] `http://z3.codeplex.com/releases/view/95640`
[3] `https://github.com/anhtrung93/Z3-str-Regex`
[4] `http://www.comp.nus.edu.sg/~trinhmt/S3/`
[5] `http://www.boost.org/users/history/version_1_57_0.html`

Z3-regex solver respectively.

1. Download the source code of Z3 4.1.1 and Z3-regex from `https://github.com/anhtrung93/Z3-str-Regex`
2. In the top level folder of Z3, build Z3
   - `autoconf`
   - `./configure`
   - `make`
   - `make a`
3. Download Boost C++ Libraries[6]
   - Refer to the "Getting Started Guide"[7] of Boost to install and build Boost Libraries. Boost is required when Z3-regex parsing the regular expression constraints.
   - Modify variable "Boost_path" in Z3_regex Makefile to indicate the Boost location
4. Build Z3-regex
   - Modify variable "Z3_path" in the Z3-regex Makefile to indicate the Z3 location.
   - `make`
5. Setup Z3-regex driver script
   - In "Z3-str.py", change the value of the variable "solver" to point to the Z3-str2 binary "str" just built.
6. Run Z3-regex
   - `Z3-str.py -f <inputFile>`
   - e.g: `$./Z3-str.py -f tests/star-001`

[6]`http://sourceforge.net/projects/boost/files/boost/1.57.0/`
[7]`www.boost.org/doc/libs/1_58_0/more/getting_started/unix-variants.html`

4.1.3 Experimental benchmarks

In this section, we describe the experiments to compare our tool, Z3-regex with S3 solver. Since Z3-regex is built on Z3-str, there is no need to compare with it. On the other hand, Z3-str team has released the a new version called Z3-str2 in April 2015[8]. Unfortunately, the benchmarks of Z3-str2 is not ready for us to redo the experiments by the time we write this thesis.

To compare Z3-regex and S3, we use 20 tests obtained from S3 website [9], AppScan Source [10] projects, which is also introduced in [8] and from Z3-regex benchmarks itself. All of these inputs have regular expression constraints in it, which is exactly what we need to evaluate.

String solvers such S3, CVC4 or Z3-str2 normally use Kaluza Benchmark Suite[6] which has the constraints generated by a JavaScript symbolic execution engine. However, this suite is only publicly available string constraint set, exclude finite regular expression membership. Therefore, we do not include Kaluza benchmark in our experiments.

4.1.4 Performace Results

	S3		Z3-regex	
	✓	✗	✓	✗
SAT	8	2	16	0
UNSAT	4	5	4	0
Timeout		1		
Total time(s)	891(**1x**)		1116(**1.25x**)	

Table 6: Overall results in comparison

The results we obtained are summarized in Table 6 and listed in details in Table 7. In Table 6 and "SAT" and "UNSAT" rows, (✗) denotes for the number of incorrect results, which is either an "UNSAT" response where the inputs have a solution or an "SAT"

[8]`sites.google.com/site/z3strsolver/publications`

[9]`http://www.comp.nus.edu.sg/~trinhmt/S3/`

[10]`http://www-03.ibm.com/software/products/en/appscan-source`

response with a wrong model, while (✓) is the rest. The total times count only the tests that both solvers return feasible solution.

According to Table 6, Z3-regex is (✓) in all the benchmarks while S3 solver only return 14/20 true answers. Overall, the total times of S3 is little less than Z3-regex (1.25x). However, it has a timeout test, even we set the timeout equal to 1000 seconds.

Input	S3			Z3-regex		
	Results	Models	Time(s)	Results	Models	Time(s)
regex-01	SAT	(✓)	0.160	SAT	(✓)	0.390
regex-02	SAT	(✓)	0.070	SAT	(✓)	0.058
regex-03	SAT	(✓)	0.054	SAT	(✓)	0.049
regex-04	UNSAT	(✓)	0.053	UNSAT	(✓)	0.073
regex-05	UNSAT	(✓)	0.057	UNSAT	(✓)	0.064
regex-06	SAT	(✓)	0.066	SAT	(✓)	0.059
regex-07	UNSAT	(✓)	0.037	UNSAT	(✓)	0.039
regex-08	SAT	(✓)	0.050	SAT	(✓)	0.044
regex-09	UNSAT	(✓)	0.039	UNSAT	(✓)	0.045
regex-10	UNSAT	(✗)	0.054	SAT	(✓)	0.057
regex-11	SAT	(✓)	0.064	SAT	(✓)	0.046
regex-12	SAT	(✗)	0.094	SAT	(✓)	0.066
regex-13	UNSAT	(✗)	0.063	SAT	(✓)	220.172
regex-14	Time out		999.99	SAT	(✓)	0.129
regex-15	SAT	(✗)	0.062	SAT	(✓)	0.068
regex-16	UNSAT	(✗)	0.058	SAT	(✓)	0.341
regex-17	UNSAT	(✗)	0.059	SAT	(✓)	1.175
regex-18	UNSAT	(✗)	0.059	SAT	(✓)	5.496
regex-19	SAT	(✓)	0.148	SAT	(✓)	0.165
regex-20	SAT	(✓)	0.094	SAT	(✓)	0.084

Table 7: Details comparison performance of S3 and Z3-regex

4.2 Evaluation

In this section, we will provide some test cases that Z3-regex can handle well while S3 solver cannot. The improvements of Z3-regex over S3 solver is evaluated as well.

regex-10: A simple test for Matches() function

Code 1: Test regex-10 of Z3-regex

```
(declare-variable x String)
(assert (= true (Matches x '[abcdefg]aa') ) )
(assert (= x "aaa") )
```

Z3-regex returns SAT result:`x : string -> "aaa"`

Code 2: Test regex-10 of S3

```
(declare-variable x String)
(assert (In x (Concat (Union "a" (Union "b" (Union "c" (Union "d"
   (Union "e" (Union "f" "g")))))) "aa")))
(assert (= x "aaa") )
```

S3 returns UNSAT.

We have Code 1 and 2 are the input constraints of Z3-regex and S3 respectively. Since S3 supports only regular expression inputs in the form of Union, Concat and Star, we have to convert the benchmarks into S3's syntax. The input constraints is to find a solution of `x`, which ends with `"aa"` and its first letter is `"a"`, `"b"`, `"c"`, `"d"`, `"e"`, `"f"` or `"g"`. It can be seen clearly from the codes that `x = "aaa"` matches with the regular expression `'[abcdefg]aa'`. This is also a feasible SAT solution.

Z3-regex support the input regex constraints as its string form, so it is not only familiar with the programmers but also no need to pre-process input constraints by hand.

regex-12: A wrong model from S3 solver

Test regex-12 is provided to test the combination of a regular expression function and a high-level string operations. The solver needs to find the solution of string variable `x` that matches with `'colou?r'` (e.g, `"colour"` or `"color"`) and ends with `"ur"`. Therefore, `"colour"` should be the answer. We suppose that S3 has faced some problems in their function `Union()` in this test.

Code 3: Test regex-12 of Z3-regex

```
(declare-variable x String)
(assert (= true (Matches x 'colou?r') ) )
(assert (= true (EndsWith x "ur") ) )
```

Z3-regex returns SAT with the model `x : string -> "colour"`

Code 4: Test regex-12 of S3

```
(declare-variable x String)
(assert (= x (Concat "colo" (Concat (Union "u" "") "r"))))
(assert (= true (EndsWith x "ur") ) )
```

S3 returns SAT with `x : string -> "colo\x7f\x7f\x7fur"`.

regex-15: Equalities between Regex functions and high-level String functions

Code 5: Test regex-15 of Z3-regex

```
(declare-variable x String)
(declare-variable n Int)
(assert (= (Star 'abc|dbcabc' n ) (Concat x "bcabc") ) )
```

Z3-regex returns `x : string -> "d"` and `n : int -> 1`.

Code 6: Test regex-15 of S3

```
(declare-variable x String)
(assert (In (Concat x "bcabc") (Star (Union "abc" "dbcabc"))) )
```

S3 returns `x : string -> "\x7f\x7f"`.

We provide this test to evaluate the membership predicates solving between a `Star()` and a `Concat()` functions with 2 variables in 2 domains string and integer. Since `Star()`

function of S3 solver does not include an integer variable, we have to convert our test regex-15 into S3's equivalence. In this test, the solvers need to find a string variable `x` that satisfy: The string concatenation of `x` and `"bcabc"` matches with regular expression `'abc|dbcabc'` after "star" `n` times. The results show that Z3-regex can solve this case perfectly while S3 cannot handle it. `x = "d"` is one solution of this test, but `x="\x7f\x7f"` is not the one.

regex-17: A test from real world problems

Code 7: Test regex-17 of Z3-regex

```
(declare-fun cookie () String)
(declare-fun cookie_part1 () String)
(declare-fun cookie_part2 () String)
(declare-fun cookie_part3 () String)
(declare-fun l1 () String)
(assert (= cookie (Concat cookie_part1 (Concat cookie_part2
   cookie_part3) ) ) )
(assert (= true (Matches cookie_part2
   '(\?|;)searchLang=[abcdefghijklmn]*')))
(assert (implies (not (= "" cookie_part3) ) (= cookie_part3 (Concat
   ";" l1) ) ) )
(assert (> (Length cookie_part2) 11) )
(assert (= cookie "expires=Thu, 18 Dec 2013 12:00:00
   UTC;searchLang=nb;domain=local;") )
```

Z3-regex returns SAT and the solution:

```
cookie : string -> "expires=Thu, 18 Dec 2013 12:00:00
   UTC;searchLang=nb;domain=local;"
cookie_part3 : string -> ";domain=local;"
cookie_part1 : string -> "expires=Thu, 18 Dec 2013 12:00:00 UTC"
cookie_part2 : string -> ";searchLang=nb"
l1 : string -> "domain=local;"
```

Code 8: Test regex-17 of S3

```
(declare-fun cookie () String)
```

```
(declare-fun cookie_part1 () String)
(declare-fun cookie_part2 () String)
(declare-fun cookie_part3 () String)
(declare-fun l1 () String)
(assert (= cookie (Concat cookie_part1 (Concat cookie_part2
   cookie_part3) ) ) )
(assert (In cookie_part2 (Concat (Concat (Union "\?" ";" )
                                          "searchLang=" )
                                 (Star (Union "a" (Union "b" (Union "c"
                                    (Union "d" (Union "e" (Union "f"
                                    (Union "g" (Union "h" (Union "i"
                                    (Union "j" (Union "k" (Union "l"
                                    (Union "m" "n") ) ) ) ) ) ) ) ) ) )
                                    ) ) )
                                 )
        )
)
(assert (implies (not (= "" cookie_part3) ) (= cookie_part3 (Concat
   ";" l1) ) ) )
(assert (> (Length cookie_part2) 11) )
(assert (= cookie "expires=Thu, 18 Dec 2013 12:00:00
   UTC;searchLang=nb;domain=local;") )
```

S3 returns UNSAT.

We take this test from AppScan Benchmark Suite, which is derived from security warning output by IBM Security AppScan Source Edition and introduced in [8]. They run AppScan on popular websites, then get the program statements traces and create the constraints. While S3 solver cannot handle these representative real-world constraints, Z3-regex can solve and return a true model for it.

Chapter 5

CONCLUSION AND FUTURE WORKS

5.1 Conclusion

The thesis provides an overview of string symbolic solving techniques in security verification of web applications and the architecture of some recently SMT-based string solvers such as Z3-str and S3 solver. Since solving string constraints is currently a hot topic and the existing solvers still have some deficiencies in solving regular expression, we develop and introduce our string solver, Z3-regex, which can solve more classes of constraints that the state-of-the-art solvers cannot. Experimental results confirm that Z3-regex can solve well these cases.

Our tool is open source so our implementation can be incorporated to the existing solvers such as S3 and Z3-str2 to increase their capabilities in handling regular expression constraints. In Z3-regex, a new Z3 sort is registered for regular expression. Therefore, Z3-regex can precisely solve equalities between regular expression functions and other expressions in different domains (e.g, string). At the moment, it supports equations over $Matches()$ and $Star()$ as well as other high-level string operations: $Concat()$, $Substring()$, $Replace()$.

5.2 Future works

Solvers are one of the main research subjects I am particularly interested in. This is not only my initial study but also the topic I would like to learn more when I pursue further study after I graduated. About Z3-regex, there are still a lot of future works I am going to do.

First, I will re-base my project on my Z3-str2 instead of Z3-str as it is now. Z3-str2 experiment results also show that there are many constraints in the benchmarks that no solvers can solve. I plan to study these cases to find problems and solutions for them.

Building more real-world benchmarks for regular expressions is also an interesting problem that can give better feedback to current solvers.

Finally, we will build a front-end web for users to use and evaluate our tool. From this, we will verify real-world applications, then make the regular expression benchmarks become richer.

References

[1] Y. Zheng, X. Zhang, and V. Ganesh, "Z3-str: a z3-based string solver for web application analysis," *The 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*, pp. 114–124, 2013.

[2] D. T. Anh and T. D. Muoi, "Z3-regex: a z3-based solver for regular expression," *Student Scientific Research Contest, University of Engineering and Technology*, 2015.

[3] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," *Technical Report, Microsoft*, 2008.

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

[5] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, , and M. D. Ernst, "Hampi: A solver for string constraints," *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA '09*, pp. 105–116, 2009.

[6] P. Saxena, D. Akhawe, F. M. Steve Hanna, S. McCamant, , and D. Song, "A symbolic execution framework for javascript," *Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP '10*, pp. 513–528, 2010.

[7] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters, "A dpll(t) theory solver for a theory of strings and regular expressions," *Proceedings of the 26th International Conference on Computer Aided Verification, CAV'14*, pp. 646–662, 2014.

[8] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, J. Dolby, and X. Zhang, "Effective search-space pruning for solvers of string equations, regular expressions and length constraints," *The 27th International Conference on Computer Aided Verification (CAV 2015)*, 2015.

[9] R. S. Boyer, B. Elspas, and K. N. Levitt, “Select–a formal system for testing and debugging programs by symbolic execution,” *Proceedings of the International Conference on Reliable Software*, pp. 234–245, 1975.

[10] J. C. King, “Symbolic execution and program testing,” *Communications of the ACM*, vol. 19, pp. 385–394, 1976.

[11] W. E. Howden, “Experiments with a symbolic evaluation system,” *Proceedings, National Computer Conference*, 1976.

[12] L. A. Clarke, “A program testing system,” *ACM 76: Proceedings of the Annual Conference*, pp. 488–491, 1976.

[13] G. Redelinghuys, “Symbolic string execution,” *Master of Science in Computer Science at the University of Stellenbosch*, 2012.

[14] S. Ranise and C. Tinelli, “The satisfiability modulo theories library (smt-lib).” www.SMT-LIB.org, 2006.

[15] D. Detlefs, G. Nelson, and J. B. Saxe, “Simplify: a theorem prover for program checking,” *Journal of the ACM (JACM)*, vol. 52, pp. 365–473, 2005.

[16] P. Hooimeijer and W. Weimer, “A decision procedure for subset constraints over regular languages,” *PLDI ’09 Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pp. 188–198, 2009.

[17] M. Emm, R. Majumdar, and K. Sen, “Dynamic test input generation for database applications,” *ISSTA ’07 Proceedings of the 2007 international symposium on Software testing and analysis*, pp. 151–162, 2007.

[18] G. Wassermann and Z. Su, “Sound and precise analysis of web applications for injection vulnerabilities,” *PLDI ’07 Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pp. 32–41, 2007.

[19] G. Wassermann and Z. Su, “Static detection of cross-site scripting vulnerabilities,” *ICSE ’08 Proceedings of the 30th international conference on Software engineering*, pp. 171–180, 2008.

[20] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise analysis of string expressions," *Proceeding SAS'03 Proceedings of the 10th international conference on Static analysis*, pp. 1–18, 2003.

[21] D. Shannon, I. Ghosh, S. Rajan, and S. Khurshid, "Efficient symbolic execution of strings for validating web applications," *DEFECTS '09: Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*, pp. 22–26, 2009.

[22] G. Li and I. Ghosh, "Pass: String solving with parameterized array and interval automaton," *9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings*, pp. 15–31, 2013.

[23] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t)," *Journal of the ACM (JACM)*, vol. 53, pp. 937–977, 2006.