Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Automatically Testing Solvers for String and Regular Expressions Constraints

Bachelor Thesis

Sebastian Kühne

April 01, 2022

Advisors: Prof. Dr. Peter Müller, Alexandra Bugariu

Department of Computer Science, ETH Zürich

Abstract

SMT solvers are used to determine the satisfiablity of formulas expressed in first-order-logic. The solvers can support typical theories such as Integers and Booleans but this work focuses on strings and regular expressions constraints. SMT solvers have many applications, such as program verification and test case generation. All applications rely on the queries to be answered correctly. Nevertheless, as formulas are highly complex, ensuring correctness is hard.

To detect incorrectly answered queries by the solvers we extend a technique, which automatically generates formulas that are either satisfiable or unsatisfiable by construction and then uses this ground truth to test the solvers. We extended the technique to also support formulas containing regular expressions, as well as formulas combining regular expressions and arrays/bitvectors.

This thesis will show, that the for regular expressions extended technique, was able to detect soundness errors in state-of-the-art solver Z3. Furthermore we generated formulas, which expose incompleteness and performance issues in solvers.

Acknowledgements

I would like to thank Alexandra Bugariu for the supervision of my thesis. The weekly meetings were always helpful and inspiring. I especially want to thank her for enthusiastically introducing me into the world of researching.

Furthermore I want to thank Prof. Dr. Peter Müller for giving me the opportunity to work in the department, Nikolaj Bjørner (one of the developers of Z3) for introducing new functionality for the java API and Mitja Kulczynski for his detailed explanations for Z3str3RE.

Contents

Chapter 1

Introduction

Satisfiability modulo theories (SMT) formulas are expressed in first-order-logic. They can support typical theories such as Integers and Bitvectors but we put our main focus on SMT formulas containing *strings and regular expressions*. For instance Figure 1.1 describes a simple SMT formulas with strings and regular expressions. The left-hand side consists of a function *string_in_lang* which takes a string *s* and a regular expression *re* as arguments. This function returns *true* if *s* is included in the *language* described by *re* and $false$ otherwise. Note that here the regular expression is a constant which describes the language $L = \{w \mid w = w_1w_2 \land w_1 \in \{\text{"a","b","c"}\} \land w_2 \in \{\text{"d","e","f"}\}\}$. On the other hand the string *s* is a free variable.

Furthermore we also looked at formulas combining arrays, bitvectors and regular expressions.

SMT formulas are either *satisfiable* or *unsatisfiable*. An SMT formula is satisfiable if there exists a *model* (an assignment of free variables such that the formula evaluates to true). In Figure 1.1 we can assign *"ad"* to *s*. Because *"ad"* is in *L*, *string_in_lang* evaluates to true and therefore the formula evaluates to true as well. Hence we have found an assignment of the free variables such that the formula evaluates to true. Thus the formula is satisfiable.

An SMT formula is unsatisfiable if there does not exist a model. The SMT formula in Figure 1.2 evaluates to false for all possible values of the free

$$string\_in\_lang(s, ["a - c"]["d" - "f"]) = true$$

Figure 1.1: Satisfiable SMT formula where s has type String.

$$string\_in\_lang(s, re\_empty) = true$$

Figure 1.2: Unsatisfiable SMT formula where s has type String and re_empty is a constant regular expression describing the empty language.

$$string\_in\_lang('''', loop(189, 0, string\_to\_regex('''')))$$

Figure 1.3: Formula that exposes a soundness bug in Z3 (version 4.8.7)

variable s as the regular expression *re_empty* which describes the empty language, does not match any strings.

SMT solvers try to determine whether a formula is satisfiable or unsatisfiable. If they return *sat* they also return a model for the formula. If they return *unsat* they optionally return a set of clauses that lead to a contradiction, called the *unsat core*. They can also time out (they cannot solve the query within a given time span). Timeouts often point to *performance issues*. Furthermore as SMT solvers support undecidable theories they can also return *unknown* if they cannot determine whether the SMT formula is satisfiable or unsatisfiable.

The following two issues can have a negative impact on SMT solvers (besides timing out):

- **Unsoundness:** An SMT solver is considered unsound if it returns sat for an unsatisfiable formula or vice versa. We also consider it unsound if it correctly returns sat but produces an *invalid model* or if it correctly returns unsat but returns an *incorrect unsat core* (a sub-formula that is satisfiable).
- **Incompleteness:** We consider an SMT solver incomplete if it returns unknown for a decidable formula.

Besides detecting soundness errors in the solvers, we were also interested in generating formulas that expose incompleteness or performance issues.

1.1 Motivation

Strings with regular expressions constraints are widely used in "find" or "find and replace" operations. Furthermore they also play a vital role in input validation. For example checking whether a date entered by a user is valid or if a password matches given constraints.

SMT Solvers such as *Z3* [13] and *CVC5* [1] are used to solve SMT formulas that can contain regular expressions. SMT solvers have many applications, such as program verification and test case generation. All applications rely on the SMT-solver to be correct.

The formula from Figure 1.3 checks if the language generated by repeating the empty string from 189 times to 0 times includes the empty string. According to the *SMT-LIB standard* [11], the *reference semantics* used by Z3 and CVC5, a repetition with lower bound higher than upper bound results in an empty language. As the empty language does not contain the empty string the SMT formula is unsatisfiable. Nevertheless the SMT solver Z3 [13] (version from April 2020) incorrectly returned sat which is a soundness issue.

Although the correctness of the solvers is crucial, it is unfortunately not guaranteed as Figure 1.3 shows. Detecting soundness errors in the solvers was a primary motivation for our thesis. As Chapter 6 shows, we were able to generate formulas that do expose soundness issues.

1.2 This work

We extended the approach used in [19], which automatically generates formulas that are either sat or unsat by construction and then use this ground truth to test the solvers, to also support formulas containing regular expressions. We extended the approach used in [18], which automatically generates unsatisfiable formulas with only constants using concrete execution, to support regular expressions.

1.3 Contributions

The main contributions of the thesis are the following:

- We extended the technique of [19] to support regular expressions.
- We extended the technique of [18] to generate unsatisfiable formulas with only constants to support formulas with regular expressions.
- We extended [19] and [18] to generate formulas combining regular expressions and arrays/bitvectors.
- We evaluated the extended technique with state-of-the art solvers Z3 [13] and CVC5 [1].

1.4 Outline

The outline of this thesis is as follows: Chapter 2 summarizes the extension of the approach of [19] and [18] to support regular expressions. In Chapter 3 we extend the technique of [19] and [18] to support formulas, combining regular expressions and bitvectors/arrays. Chapter 4 discusses technical details that were needed for extending the approaches. In Chapter 5 we will discuss the implementation needed for the techniques. In Chapter 6 we will evaluate the extended technique. In Chapter 7 we present related work. Chapter 8 draws conclusions of our work and presents future work.

Chapter 2

Overview

We started from the technique described in [19] which automatically generates SMT formulas from the string theory. The basic approach of [19] to construct and transform formulas is:

Step 1. Generate simple formulas which are either sat or unsat by construction.

Step 2. Apply automatic transformations to these formulas to obtain more complex, *equisatisfiable* formulas.

The ground truth of the formulas is known as the formulas are either satisfiable or unsatisfiable by construction. These formulas can then be used as inputs to test SMT solvers.

We extended the technique proposed in [19] to also support regular expressions. Table 2.1 shows all considered operations involving regular expressions.

Table 2.1: Regular expression operations, grouped by their return type

Return type	Operation		
Regular expression	$all()$	$all\_char()$	$none()$
	$string\_to\_regex(s)$	$concat(re_1, re_2)$	$union(re_1, re_2)$
	$intersection(re_1, re_2)$	$difference(re_1, re_2)$	$complement(re)$
	$kleene\_closure(re)$	$kleene\_cross(re)$	$option(re)$
	$range(s_1, s_2)$	$power(n, re)$	$loop(n_1, n_2, re)$
String	$re\_replace(s, re, t)$	$re\_replace\_all(s, re, t)$	
Boolean	$string\_in\_lang(s, re)$		

s, s_1, s_2, t: type String re, re_1, re_2: type Regular expression
n, n_1, n_2: type Int

Table 2.2: String operations, grouped by their return type

Return type	Operation		
String	$at(s, off)^*$	$concat(s, t)$	$intToStr(n)$
	$replace(s, t, u)$	$substr(s, off, len)$	
Integer	$indexOf(s, t, off)$	$length(s)$	$strToInt(s)$
Boolean	$contains(s, t)$	$equals(s, t)$	$prefixOf(s, t)$
	$sufixOf$(s,t)		

* returns a char; s, t, u: type String; n, off, len: type Integer

$$a = (re\_empty\_string)a$$

Figure 2.1: Regex equality where re_empty_string **is a constant regular expressions describing the language containing only the empty string.**

$$range(s_1, s_2) = res$$

Figure 2.2: Satisfiable formula where s_1, s_2 **have type String and** res **has type Regular expression.**

2.1 Regular expression equality

Concretely performing regex equality is essential for most transformations in this paper. According to SMT-LIB *regular expressions are equal if they describe the same language*. As a consequence the regex equality in Figure 2.1 holds, although the string representation of the two regular expressions in the Figure is different, because both regex describe the language containing only String "a".

We use the *dk.brics.automaton* [2] to represent regular expressions. Dk.brics.automaton transforms regex into automatons and thus allowing us to compare the language of regular expressions, which enables us to perform correct regular expression equality checks.

2.2 Generating formulas

We follow the approach from [19] and generate satisfiable and unsatisfiable formulas separately. In the following sections we present how we generate satisfiable and unsatisfiable formulas containing regular expression operations.

2.2.1 Generating satisfiable formulas

In step 1 we take formulas involving an operation with regular expressions described in Table 2.1 with unconstrained parameters and an unconstrained result. These formulas are trivially satisfiable as these operations are total functions. For instance in Figure 2.2 we use the *range* function which takes

Table 2.3: Equivalent formulas for regex operations

Id	Regex operation	Equivalent formula
E1	$difference(re_1, re_2) = res$	$res = re_1 \cap re_2^c$
E2	$kleene\_cross(re) = res$	$res = re\{++\}_r re^*$
E3	$option(re) = res$	$res = re\_empty\_string \cup re$
E4	$power(n, re) = res$	$(res = re\_empty\_string \textbf{ if } n = 0) \wedge (res = re\{++\}_r power(n-1, re) \textbf{ if } n > 0)$
E5	$loop(n_1, n_2, re) = res$	$(res = re\_empty \textbf{ if } n_1 > n_2) \wedge (res = power(n_1, re) \textbf{ if } n_1 = n_2) \wedge (res = power(n_1, re) \cup .... \cup power(n_2, re) \textbf{ if } n_1 < n_2)$
E5′	$loop(n_1, n_2, re) = res$	$(res = re\_empty \textbf{ if } n_1 > n_2) \wedge (res = power(n_1, re) \textbf{ if } n_1 = n_2) \wedge (res = power(n_2, re) \cup loop(n_1, n_2 - 1, re) \textbf{ if } n_1 < n_2)$
E6	$range(s_1, s_2) = res$	$(res = string\_to\_regex(s_1) \cup .... \cup string\_to\_regex(s_2) \textbf{ if } \lvert s_1 \rvert = \lvert s_2 \rvert = 1) \wedge (res = re\_empty \textbf{ if } \lvert s_1 \rvert \neq 1 \vee \lvert s_2 \rvert \neq \vee s_1 > s_2)$
E7	$complement(re) = res$	$res = difference(re^c, re)$

s_1, s_2 : type String; re, re_1, re_2; type Regular expression; n, n_1, n_2 : type Integer;
re_2^c denotes the complement of re_2; $\{++\}_r$ denotes regex concatenation;

two strings s_1, s_2 as arguments and returns the language containing all string of length 1 between s_1, s_2 if s_1 and s_2 are single characters (ordered by their corresponding ASCII value) and the empty language otherwise. Both arguments and the results are unconstrained.

In step 2 we apply one of the extended satisfiability preserving transformations. These transformations are described in detail in Section 2.3.

2.2.2 Generating unsatisfiable formulas

[19] considers equivalent formulas A and B and defines a formula $F := \neg A \wedge B$, which is unsatisfiable by construction as $A \equiv B$. In a second step it transforms F into a more complex formula with a larger unsat core. After applying a transformation, the larger unsat core is unique and known by construction.

Analogously to the solution from [19] for Strings, we built Table 2.3 listing equivalent formulas, that we considered. The table is based on SMT-LIB's [11] description of the operations involving regular expressions. Having built Table 2.3, allows us to generate formulas $F := \neg A \wedge B$, where $A \equiv B$ with regular expressions.

Table 2.4: Set of predefined constants used in our experiments

Pool type	Elements				
String	*""*	*"a"*	*"b"*		
Integer	0	1	3	4	
Regular Expression	a	b	*re_empty*	*re_all_char*	*re_all*

a and *b* are the regular expressions describing the language containing string *"a"* and string *"b"* respectively; *re_empty* describes the empty language; *re_allchar* describes the language containing all strings of length 1; *re_all* describes the language containing all strings

2.3 Transforming satisfiable formulas

In this Section we present two satisfiability preserving transformations, which were first introduced for the String theory in [19]. We extended the transformations for regular expressions.

Both transformations rely on *concrete execution*. [12] implements the string operations from Table 2.2 according to the SMT-LIB standard.

To support regular expressions we extended the existing implementation. We carefully implemented the operations from Table 2.1 according to the reference semantics of SMT-LIB, to have an executable version of the regex operations.

Furthermore our transformations rely on a pool of predefined constants. Table 2.4 shows the constants of the corresponding types used in our experiments.

2.3.1 Constant assignment transformation

[19] describes the Constant assignment transformation as inspired by boundary testing, because many software errors are caused by the incorrect handling of corner cases.

Having built an executable semantics for the operations from Table 2.1, extending constant assignment transformation for regular expressions was straightforward. The transformation for regular expressions proceeds as follows:

We take any operation from Table 2.1 with some arguments from the corresponding pools from Table 2.4. We then obtain a result by concretely executing this operation. We then build the satisfiable formula by setting the operation with the corresponding arguments equal to the the obtained result. Lastly we replace some of the constants by free variables. This formula is still satisfiable, because assigning the same constants to the corresponding replaced variables is a model for the formula. This process is done exhaustively for all possible initial values from the pools, we also exhaustively consider

$$range(''b'',''a'') = re\_empty$$

Figure 2.3: Satisfiable formula obtained by concretely executing the *range* function with arguments *"b"* and *"a"*.

$$range(''b'', tmp\_str1) = tmp\_regex2$$

Figure 2.4: Satisfiable formula obtained by constant assignment transformation, where tmp_str1 has type String and tmp_regex2 has type Regular expression.

all possible replacements of constants. Let us illustrate this technique on an example. In Figure 2.3 we concretely execute the range operation with arguments *"b"* and *"a"*. As the ASCII representation of *"b"* is greater than the ASCII representation of *"a"* the result of the concrete execution yields re_empty. For the formula from Figure 2.3 Z3_seq [13] (version 4.8.14) and Z3str3RE [17] incorrectly returned unsat. The bug was confirmed by the developers from Z3str3RE.

In the second step we replace some of the constant values by free variables in the formula. For example we can replace the constants *"a"* and re_empty by free variables of the corresponding type, shown in Figure 2.4. This formula is satisfiable as we can assign *"a"* to s and re_empty to res which is a model for the formula. Hence we have generated a satisfiable formula with a known model.

2.3.2 Term synthesis transformation

[19] describes the goal of term synthesis to test the interactions between different operations. This is in contrast to constant assignment transformation, where each operation was tested in isolation.

Term synthesis transformation for regular expressions proceeds as follows. We exhaustively evaluate all operations from Table 2.1 and Table 2.2 with arguments from the pools, building a *term pool*. Analogously to constant assignment transformation we concretely execute an operation from Table 2.1 with arguments from the corresponding pools (Table 2.4) building an equality. We then replace the result and the arguments with other terms from the term pool with equal results. Note that we rely on regex equality as shown in 2.1 when we replace regex constants by regex terms. Finally we replace all constants in the formula by free variables. Note that equal constants get replaced by the same free variable to obtain a more tight formula. Term synthesis is done exhaustively for all possible terms.

To give further insight to the technique we illustrate term synthesis transformation on an example. First we exhaustively evaluate all operations from Table 2.1 and Table 2.2 with arguments from the pools, building the term pool, shown in Table 2.5. We concretely execute the $string\_in\_lang$

$$string\_in\_lang(at(tmp\_str0, tmp\_int1), range(tmp\_str0, tmp\_str0)) = contains(tmp\_str0, tmp\_str0)$$

Figure 2.5: Satisfiable formula obtained by term synthesis transformation where tmp_str0 has type String and tmp_int1 type int. Z3str3RE returns unsat.

Table 2.5: Term pool

Term	Result
....	....
$at(''a'', 0)$	$''a''$
$range(''a'', ''a'')$	a
$contains(''a'', ''a'')$	$true$
....	....

Term pool obtained by concretely executing operations from Table 2.1 and Table 2.2 with constant arguments.

operation with String $''a''$ and regular expression a. The result of the operation is trivially true. This equality builds our simple formula. Next we replace the constants by terms from Table 2.5 which evaluate to the same constant. We replace $''a''$ by $at(''a'', 0)$ because the result of $at(''a'', 0)$ is $''a''$, replace a by $range(''a'', ''a'')$ ($range(''a'', ''a'') = a$) and replace the result by $contains(''a'', ''a'')$ ($contains(''a'', ''a'') = true$.

Finally we replace the constants by free variables, note that we assign the same variable to the same constants. This particular execution of term synthesis results in the formula from Figure 2.5. Note that the formula is still satisfiable as assigning the constants, that were replaced by the free variables, to the same variables is a model for the formula.

Z3str3RE [17] returned unsat for this formula. The bug was confirmed and fixed by the developers from Z3str3RE.

2.3.3 Term synthesis without regular expression variables

SMT solvers cannot handle formulas well with regular expression variables, which we show in Chapter 6. This was the motivation for a slightly modified term synthesis, involving no variables of type regular expression.

We define our regular expression pool to consist of regular expressions only describing a language containing a single string.

The only modification to term synthesis as described in Section 2.3.2, is in the constant replacement process. Constant regular expressions that only contain a String s, first get rewritten by $string\_to\_regex(s)$ and instead of replacing the regular expression by a constant we replace the string s by a string variable.

$$concat(loop(1,1,a), string\_to\_regex("")) = range("a","a")$$

Figure 2.6: Satisfiable formula obtained by term synthesis transformation before replacing constants.

$$concat(loop(1,1,tmp\_regex0), string\_to\_regex(tmp\_str0) = range(tmp\_str\_1, tmp\_str1)$$

Figure 2.7: Satisfiable formula obtained by Term synthesis transformation with regular expressions. tmp_regex0 **has type regular expression,** tmp_str0 **and** tmp_str1 **have type String. Note that the loop operation supports no integer variables. They have to be constant.**

$$concat(loop(1,1,string\_to\_regex(tmp\_str1)), string\_to\_regex(tmp\_str0) = range(tmp\_str\_1, tmp\_str1)$$

Figure 2.8: Satisfiable formula obtained by term synthesis transformation without regular expression variables (2.3.3). tmp_str0 **and** tmp_str1 **have type String. Note that the loop operation supports no integer variables. They have to be constant. Z3_ seq (version 4.8.14) returned an invalid model.**

In our experiments we define the regular expression pool to consist of regex a and b. By construction of the regex pool all constant regular expression occurring before replacement, only contain a string. Note that we could also include more complex constant regex, that cannot be rewritten using the *string_to_regex* function in our pool and simply leave them constant in our synthesized formula.

Let us illustrate this slight modification in the replacement process on an example. Both transformations obtain the same initial formula from Figure 2.6. The difference between the two transformations is that term synthesis with regular expressions (Section 2.3.2) replaces a by a variable tmp_regex0 of type regular expression. Whereas the slightly modified technique (Section 2.3.3) replaces a by $string\_to\_regex(tmp\_str1)$. Term synthesis with regular expression variables obtains formula Figure 2.7, whereas the term synthesis without regular expressions, obtains Figure 2.8.

Z3_ seq [13] (version 4.8.14) returned an invalid model for the formula from Figure 2.8.

2.4 Transforming unsatisfiable formulas

In the unsatisfiability preserving transformations introduced in [19], one considers formulas $F(x,y) := \neg A(x) \wedge B(x,y)$. Note that the variable x appears in both A and B. The basic idea is to introduce a fresh free variable x_{fresh}, that replaces all occurrences of x in $B(x,y)$ and then to conjoin a new clause $C(x, x_{fresh})$ that implies that $x = x_{fresh}$. Note that the solvers need to perform additional reasoning steps, as they also have to include the additional clause C to prove its unsatisfiability. Thus the unsat core is

Table 2.6: Equalities between a regular expression operation and non-constant and constant regular expressions

Id	Equality
NC1	$union(re, re\_empty) = re$
NC2	$union(re, re) = re$
NC3	$intersection(re, re\_all) = re$
NC4	$intersection(re, re) = re$
NC5	$difference(re, re\_empty) = re$
NC6	$concat(re, re\_empty\_string) = re$
NC7	$concat(re\_empty\_string, re) = re$
NC8	$power(1, re) = re$
NC9	$loop(1, 1, re) = re$
NC10	$replace\_re(s_1, re, s_2) = s_1$ **if** $\forall w, w_1, w_2\ (s_1 = w_1\{++\}_s w\{++\}_s w_2) \implies string\_in\_lang(w, re) = false$
NC11	$replace\_re\_all(s_1, re, s_2) = re$ **if** $\forall w, w_1, w_2\ (s_1 = w_1\{++\}_s w\{++\}_s w_2) \implies string\_in\_lang(w, re) = false$
C1	$string\_in\_lang(s, re\_all) = true$
C2	$string\_in\_lang(s, re\_empty) = false$
C3	$range(s_1, s_2) = re\_empty$ **if** $(\lvert s_1\rvert \neq 1) \vee (\lvert s_2\rvert \neq 1) \vee (s_1 > s_2)$
C4	$power(0, re) = re\_emtpy\_string$
C5	$loop(n_1, n_2, re) = re\_empty$ **if** $n_1 > n_2$
C6	$union(re, re^c) = re\_all$
C7	$intersection(re, re^c) = re\_empty$

s_1, s_2, s, w, w_1, w_2: type String; re, re_1, re_2: type Regular expression; n, n_1, n_2: type Int; $\{++\}_s$ denotes string concatenation

$$\neg(kleene\_cross(re) = res) \wedge (res = re_{fresh}\{++\}_r re^*_{fresh}) \wedge (union(re_{fresh}, re\_empty) = re)$$

Figure 2.9: Unsatisfiable formula obtained by variable replacement transformation. All variables are of type regular expression. Z3_seq 4.8.14 returned unknown.

extended by C.

Analogously to Table 3 in [19], which describes equalities between string operations and non-constant and constant strings, we designed equalities between regex operations and constant/non-constant regex, shown in Table 2.3. These equalities are used to construct the new clause C.

In the following subsections we illustrate two unsatisfiability preserving transformations that were extended for regular expressions.

2.4.1 Variable replacement transformation

Variable replacement transformation proceeds as follows: We choose any equality from Table 2.3 and use it to construct our formulas A and B. We choose an identity NC1-NC11 from Table 2.6 for the additional clause.

$$\neg(option(re) = res) \wedge (res = z_{fresh} \cup re) \wedge (power(0, re) = z_{fresh})$$

Figure 2.10: Unsatisfiable formula obtained by constant replacement transformation. All variables are of type regular expression.

To give more insight to the technique we illustrate it on an example: We use $E2$ from Table 2.3 to build the formulas A and B. Thus $A := (kleene\_cross(re) = res)$ and $B := (res = re\{++\}_r re^*)$. A is equivalent to B, as the $kleene\_cross(re)$ operation is defined as the concatenation of re and the Kleene star of re, according to SMT-LIB. In the first step we negate A and replace all occurrences of re in B by re_{fresh}. In the second step we need to add an additional clause that implies $re = re_{fresh}$. We can for instance choose $NC1$ from Table 2.6. Thus we define $C := union(re_{fresh}, re\_empty) = re$. Finally we conjoin $\neg A$, $B[re\_fresh/re]$ and C which results in the unsatisfiable formula from Figure 2.9.

Z3_seq 4.8.14 returned unknown for this formula, exposing incompleteness.

2.4.2 Constant replacement transformation

In constant replacement transformation we consider formulas $F(x, y) := \neg A(x) \wedge B(c, y)$, where c is a constant, chosen from Table 2.3. Instead of replacing a variable in B as done in variable replacement transformation (Section 2.4.1), we replace a constant c by a fresh free variable z_{fresh}. We then add a new clause C chosen from Table 2.6 C1-C7, that implies $z_{fresh} = c$.

Let us illustrate this transformation on an example. We consider $\neg A := \neg(option(re) = res)$ and $B := (res = re\_empty\_string \cup re)$ (E2 from Table 2.3). In the first step we replace the constant re_empty_string in B by z_{fresh}. In the second step we use C4 from Table 2.6 to define $C := power(0, re) = z_{fresh}$. Note that according to SMT-LIB, $power(0, re)$ is equal to re_empty_string. Thus C implies $z_{fresh} = re\_empty\_string$. We can now conjoin all 3 clauses and obtain the formula from Figure 2.10. Note that all 3 clauses are needed to prove its unsatisfiability.

2.5 Unsatisfiable formulas with only constants

In this section we consider an approach for generating unsatisfiable formulas with only constants using concrete execution. We extend Olivier Becker's approach presented in [18] for transforming unsat formulas, only involving constants, to also support regular expressions. Having built the concrete execution for regular expressions, extending the transformations to support regular expressions was straightforward.

$$range("a","b") = re\_empty$$

Figure 2.11: Unsatisfiable formula obtained by transforming unsatisfiable formulas with constant values.

2.5.1 Transforming unsatisfiable formulas with constant values

Similar to constant assignment transformation (Section 2.3.1), we concretely execute operations from Table 2.1 with arguments from the corresponding pools (Table 2.4). Finally we set the operation equal to some constant from the corresponding pool not equal to the result of the concretely executed operation. Note that we need to perform regex equality on a language level, as shown in Section 2.1. We do this procedure exhaustively for all possible results and arguments.

We illustrate this technique on one of the generated formulas. We concretely execute the *range* function with constant *"a"* as the first argument and constant *"b"* as the second argument. This results in the regular expression, describing the language containing strings *"a"* and *"b"*.

We then set $range("a","b")$ equal to a regex from the pool, describing a different language. We can for instance choose *re_empty* which results in the unsatisfiable formula from Figure 2.11.

Note that unlike in constant assignment transformation, we cannot substitute the constants by free variables. Let us assume we substitute *"a"* by a free variable *tmp_str*0. Although we know that if we assign *"a"* to *tmp_str*0 the formula will evaluate to false, the formula has to be false for all assignments in order to be unsat. Assigning *"c"* to *tmp_str*0 is a model for the formula. Thus replacing *"a"* by *tmp_str*0 makes the formula satisfiable. This unsound replacement was not done in [18] and neither by us, but illustrates why we have to use a different procedure for transforming unsatisfiable formulas with variables.

2.5.2 Transforming unsatisfiable formulas with constant values and complex result

Transforming unsatisfiable formulas with constant values and complex result proceeds analogously to Transforming unsatisfiable formulas with constant values (Section 2.5.1) for the first step.

The main difference is that we set the operation equal to some more complex term with a result not equal to the concretely executed operation's result. The terms are chosen from the term pool which is built analogously to the term pool from term synthesis (Section 2.3.2).

One formula that was generated by this transformation is shown in Figure 2.12.

$$range(''a'', ''b'') = concat(a, b))$$

Figure 2.12: Unsatisfiable formula obtained by transforming unsatisfiable formulas with constant values and complex result, where $''a''$ and $''b''$ are String constants and a, b are constant regex.

Chapter 3

Combining regular expressions with arrays and bitvectors

In this chapter we consider formulas combining multiple theories. We look at formulas containing combinations of regular expressions and bitvectors [10], arrays [9]. Table 3.2 describes all the operations grouped by their return type including bitvector and array operations.

The main interaction between the three theories happens with Arrays using index or element type regular expression. Note that in contrast to for instance java, arrays in SMT-LIB do not have to be of index type Integer. In Figure 3.1 we create an array with index type *Regular Expression* and element type *Bitvector*.

3.1 Adapting the technique

In the following sections we extend term synthesis and unsat formulas with constants transformations to support combination of theories. Both transformations rely on concrete execution. Becker [18] implemented all operations containing bitvectors or arrays from Table 3.2 according to the SMT-LIB semantics. However he did not consider arrays having index or element type regular expression. In order to have an executable, we extended his implementation to also support arrays with index or element type regular expression.

$$Array(Regular\ Expression, Bitvector)$$

Figure 3.1: Array with index type regular expression and element type Bitvector

Table 3.1: Bitvector pool

$x\#0$	$x\#1$

Both bitvectors have fixed size 4.

Table 3.2: Considered operations grouped by their return type

Return type	Operation		
Regular expression	$all()$	$all\_char()$	$none()$
	$string\_to\_regex(s)$	$concat(re_1, re_2)$	$union(re_1, re_2)$
	$intersection(re_1, re_2)$	$difference(re_1, re_2)$	$complement(re)$
	$kleene\_closure(re)$	$kleene\_cross(re)$	$option(re)$
	$range(s_1, s_2)$	$power(n, re)$	$loop(n_1, n_2, re)$
String	$re\_replace(s, re, t)$	$re\_replace\_all(s, re, t)$	$at(s, off)$
	$concat(s, t)$	$intToStr(n)$	$replace(s, t, u)$
	$substr(s, off, len)$		
Boolean	$string\_in\_lang(s, re)$	$contains(s, t)$	$prefixOf(s, t)$
	$equals(s, t)$	$bvult(v, w)$	
Integer	$indexOf(s, t, off)$	$length(s)$	$strToInt(s)$
	$bv2nat(v)$		
Bitvector	$concat(v, w)$	$extract(a, b, v)$	$bvnot(v)$
	$bvand(v, w)$	$bvor(v, w)$	$bvadd(v, w)$
	$bvmul(v, w)$	$bvudiv(v, w)$	$bvurem(v, w)$
	$bvshl(v, w)$	$bvlshr(v, w)$	
Array	$store(array(T_1, T_2), i, e)$		
T_2	$select(array(T_1, T_2), i)$		

T_1, T_2 : Regular expression, String, Boolean, Integer or Bitvector; s, s_1, s_2, u, t: type String re, re_1, re_2: type Regular expression;
n, n_1, n_2, a, b, off, len: type Integer; v, w: type Bitvector of length m where m is a strictly positive integer; i: type T_1; e: type T_2

As we are now also considering operations from the array and bitvector theory, we need to set up a pool of predefined constants for arrays and bitvectors, as done for Regular expressions, Strings and Integers in Table 2.4.

The considered bitvectors are listed in Table 3.1. As we are interested in the combination of regular expressions and bitvectors/arrays we restrict our array pool to consist of arrays having either index or value type regular expression. Arrays in SMT-LIB also need to have an initial value type, that is why we consider all possible initial values from our other pools as initial values for the arrays.

Furthermore [18] tested the interaction between arrays, bitvectors and string operations. That is why we explicitly only generate formulas that contain both regular expressions and an array or a bitvector.

3.1.1 Term synthesis

For combined term synthesis we follow the approach from Section 2.3.2. We now also consider array and bitvector operations and also include a bitvector and array pool.

Figure 3.2 shows one automatically generated satisfiable formula by term

$$select(store(tmp\_array0, tmp\_regex1, tmp\_bv3), kleene\_cross(\\ select(tmp\_array6, tmp\_int7))) = bvadd(tmp\_bv10, tmp\_bv3)$$

Figure 3.2: Satisfiable formula obtained by term synthesis. tmp_array0 : **type Array(Regex, Bitvector);** tmp_array6 : **type Array(Int,Regex),** tmp_regex1 : **type Regex;** tmp_int7 : **type Integer;** tmp_bv10, tmp_bv3: **type Bitvector; all bitvectors have fixed size 4.**

$$bvult(x\#1, x\#1) = select((as\ const\ (Array\ Regex\ Bool)\ true), all\_char)$$

Figure 3.3: Unsatisfiable formula obtained by unsatisfiable formulas with constant values and complex result combining different theories.

synthesis. Z3_seq 4.8.14 correctly returned sat with a correct model for this formula.

3.2 Unsatisfiable formulas with constants and complex result

We followed the same approach from Section 2.5.2 , but we now also consider array and bitvector operations and also include a bitvector and array pool. One automatically generated formula by the technique is shown in Figure 3.3.

Chapter 4

Technical details

In this chapter we explore some additional technical details for generating formulas with regular expressions.

4.1 Regular expression pool

We decided to fix the regular expression pool in our experiments, mainly because we wanted to be exhaustive. An alternative approach is to randomly create a regular expression pool, which is presented in the following section.

4.1.1 Randomized regular expression pool

The following procedure relies on concrete execution and generates arbitrarily many constant regular expressions given a fixed String pool *string_pool* and Integer pool *integer_pool* :

1. Initialize an empty *regular_expression_pool*.
2. Add *re_all*, *re_empty*, *re_allchar* to the *regular_expression_pool*
3. Add evaluated $string\_to\_regex(s)$ to *regular_expression_pool* for all s in *string_pool*
4. Randomly choose operations from Table 2.1 with return type Regular Expression, concretely execute the operations with randomly chosen arguments from the corresponding pools and add the result to the pool. For instance we evaluate $concat(re_1, re_2)$ where re_1, re_2 are some randomly chosen constant regular expressions of the pool and add the result to the pool.
5. Repeat step 5 until a certain size of the pool is reached.

$$concat(string\_to\_regex("a"), string\_to\_regex("b")) =$$
$$concat(string\_to\_regex("a"), string\_to\_regex("b"))$$

Figure 4.1: Regex operation and result.

$$concat(string\_to\_regex("a"), string\_to\_regex("b")) = string\_to\_regex("ab")$$

Figure 4.2: Regex operation and simplified result.

4.2 Representing regex in SMT-LIB

The only way of representing regex in SMT-LIB is by using the functions described in Table 2.1 with return type Regular Expression. As a consequence the result obtained by some operation with return type Regular expression can only be expressed as an other regex operation.

This in contrast to operations with return type String, where the result can simply be represented as a string. Note that the representation of the string result is unique, whereas there are infinitely many ways of representing a result of type Regular expressions (different operations, describing the same language).

Let us consider Figure 4.1 where we represent the result of the operation simply as the operation. Nevertheless we probably prefer the representation of the result used in Figure 4.2. This simplification also leads to more interesting test cases for the solvers (in constant assignment transformation from Section 2.3.1), as the solver cannot simply compare the string representation of the operations to reason about regex equality.

In the following section we present some simplifications for regex.

4.2.1 Regex simplifications

Table 4.1 shows all considered simplifications. These simplifications are applied for regex results obtained by concrete executions in constant assignment transformation (Section 2.3.1). Note that there might exist more possible simplification rules.

Table 4.1: Regex simplifications

Initial regex	**Simplified regex**
$concat(string\_to\_regex(s_1), string\_to\_regex(s_2))$	$string\_to\_regex(s_1\{++\}_s s_2)$
$union(re_1, re_2)$, where $re_1 = none()$	re_2
$union(re_1, re_2)$, where $re_2 = none()$	re_1
$union(re_1, re_2$, where $re_1 = re_2$	re_1
$intersection(re_1, re_2)$, where $re_1 = all()$	re_2
$intersection(re_1, re_2)$, where $re_2 = all()$	re_1
$intersection(re_1, re_2)$, where $re_1 = re_2$	re_1
$range(s_1, s_2)$, where $s_1 = s_2$ and $\|s_1\| = \|s_2\| = 1$	$string\_to\_regex(s_1)$
$power(1, re)$	re
$loop(1, 1, re)$	re
re, where $re = none()$	$none()$
re, where $re = all()$	$all()$

re, re_1, re_2 : type Regular expression; s_1, s_2: type String.

Chapter 5

Implementation

We extended the existing implementation [12] to support regular expressions. Furthermore we also integrated the code from [18] for bitvectors/arrays and extended the arrays to support index and value type Regular expression.

We used Java JDK 11.0.14 [6] and Z3 (4.8.14) Java API [15] for the implementation.

5.1 Structure

Our code is divided into two parts: *generator* and *runner*. In the generator we generate formulas which are either satisfiable or unsatisfiable by construction. If the formulas are satisfiable, we additionally provide a model for the formula and if the formula is unsatisfiable we provide an unsat core.

In the runner we run the generated tests with some SMT-solver and then validate the output of the solver. If the solver returns sat we additionally validate its model. If the solver returns unsat we check its unsat core.

Becker documented the existing implementation for String operations as well as Arrays and Bitvectors in [18], which is why we put our primary focus on describing newly added functionality involving regular expressions.

5.2 Reglan types

In Section 2.1 we have seen that representing regex as strings is not sufficient. We use dk.brics.automaton [2] to represent regex. This package also supports non standard representation of regular expressions, such as intersection and complement. Furthermore dk.brics.automaton automaton package transforms regex into automatons. This allows us to perform equality checks on regex. Note that performing equality checks and including complement/in-

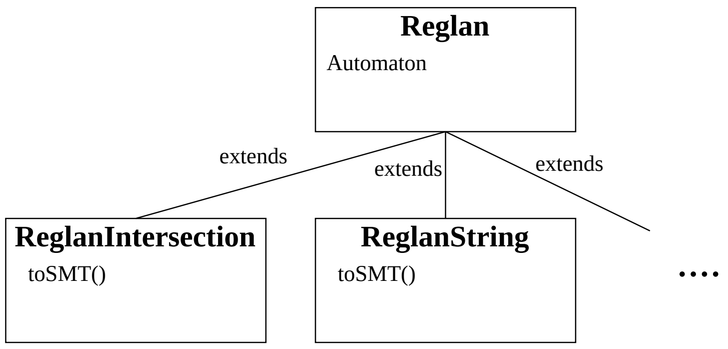

Figure 5.1: Reglan inheritance

tersection operation for regex is not included in widely used regex packages such as *java.util.regex* [7].

Dk.brics.automaton [2] does not support transformation to SMT formulas. As this SMT transformation was essential, we needed to write a wrapper for dk.brics.automaton to support conversion to SMT.

The wrapper class Reglan (Listing 5.1) has a String field *regVal* which is the String value used to build an object of type RegExp *regexp*. RegVal needs to carefully follow the semantics of [3]. With the specified regexp we build the corresponding *automaton*. This automaton allows us to use method *isStringInLang(String s)* which checks whether s is in the language described by the regex. Furthermore having built the automaton, we can perform equality checks on objects of type Reglan using the *equals* method.

For each regex type such as *ReglanIntersection* and *ReglanString* shown in Listing 5.3 and Listing 5.2 we need a class that extends the Reglan class (inheritance diagram shown in Figure 5.2) in order to track the construction history. Each subclass of Reglan implements a *toSMT()* method, allowing us to generate SMT regex. Listing 5.4 gives an example on how to use Reglan objects.

Listing 5.1: Simplified version of reglan class

```
public abstract class Reglan {
//fields needed for tracking "history"
//fields are initialized by subclasses
   protected Reglan re1;
   protected Reglan re2;
   protected int n1;
   protected int n2;
   protected String s1;
   protected String s2;
   //string regVal needed to initialize RegExp
   protected String regVal;
   //objects from dk.brics.automaton
   protected RegExp regexp;
   protected Automaton automaton;


   //returns an object of Z3 for creating SMT regex
   public abstract ReExpr toSMT(Context ctx);

   public boolean equals(Object other) {
      Reglan otherObject = (Reglan) other;
      Automaton otherAutomaton = otherObject.automaton;
      return this.automaton.equals(otherAutomaton);
      // returns true if both automatons describe same language
   }

   public boolean isStringInLang(String s) {
      // transform regex into automaton and run the automaton on s
      return automaton.run(s);

   }

}
```

Listing 5.2: Simplified version of ReglanIntersection

```
public class ReglanIntersection extends Reglan {

   public ReglanIntersection(Reglan re1, Reglan re2) {
      super(re1, re2);
      //build regVal according to dk.brics.automaton.regexp semantics
      //regVal is then used to initialize the regexp object
      this.regVal="("+re1.regVal +"&"+ re2.regVal+")";

   }

   public ReExpr toSMT(Context ctx) {
      ReExpr first = re1.toSMT(ctx);
      ReExpr second = re2.toSMT(ctx);
      ReExpr[] arr = new ReExpr[] { first, second };
      return ctx.mkIntersect(arr);
   }

}
```

Listing 5.3: Simplified version of ReglanString

```
public class ReglanString extends Reglan {
   public ReglanString(String s1) {
      super(s1);
```

```
        this.regVal = "("+s1+")";
    }


    public ReExpr toSMT(Context ctx) {
        SeqExpr seqi = ctx.mkString(s1);
        return ctx.mkToRe(seqi);

    }

}
```

Listing 5.4: Code example for Reglan

```
//build regex1=a++b
Reglan regex1 = new ReglanConcat(new ReglanString("a"), new
    ReglanString("b"));
//build regex2=(re_all_char || a)^c
Reglan regex2 = new ReglanComplement(new ReglanUnion(new
    ReglanAllChar(), new ReglanString("a")));
//perform equality check
regex1.equals(regex2);
//check whether the string "ab" is in regex1
regex1.isStringInLang("ab");
//transform regex2 to SMT
regex2.toSMT();
```

5.3 SMT-LIB reference semantics

An executable version of the SMT-LIB semantics is needed in both runner and generator. Becker [18] documents the executable semantics for string operations, array operations and bitvector operations. We present in the following sections the newly added *SMTReglan* class shown in Listing 5.5 as well as operations involving regex.

5.3.1 SMTReglan

In contrast to other SMT objects such as *SMTString* where we represent the constant value with a *value* field, we represent the constant value of SMTReglan (shown in Listing 5.5) with a *regVal* field of type Reglan (Listing 5.1). Using the objects of type Reglan to represent constant regex, allows us to check for equality on regex as described in Section 2.1. Furthermore this class allows us to represent regex variables.

Listing 5.5: Fragment of SMTReglan

```
public class SMTReglan extends SMTConstructedObject {
protected Reglan regVal;
```

```
   public SMTReglan(Reglan regVal, Operation history, boolean
       isConstant) {
   //set history, isConstant of object
      super(history, isConstant);
      //set regVal
      this.regVal = regVal;
   }

}
```

5.3.2 SMTReglan operations

SMTReglan operations follow the same pattern. One checks whether the arguments contain variables. If there are variables, the result is also a variable of the corresponding type. If the operation only involves constants, one builds the result of type Reglan, following the SMT-LIB semantics and then returns the new SMTReglan object. An example is shown in Listing 5.6 for the range operation.

Listing 5.6: Range operation simplified

```
public class ReRange extends ReglanOperation {

   public SMTConstructedObject apply() {
   //if one argument contains a variable return a new unconstrained
       SMTReglan
      if (atLeastOneUnconstrainedArgument()) {
         return new SMTReglan(unconstrained, this);
      }
      //all arguments are constants
      String s1 = arguments[0].getValue();
      String s2 = arguments[1].getValue();
      if (s1.compareTo(s2) > 0 || s1.length() != 1 || s2.length() !=
          1) {
         Reglan reglan = new ReglanEmpty();
         return new SMTReglan(reglan, this, true);
      }
      // s1<=s2
      //result of type Reglan
      Reglan reglan = new ReglanRange(s1,s2);
      return new SMTReglan(reglan, this, true);
   }
```

5.4 Generator and Runner for regular expressions

Having built the SMTReglan class and having implemented the regex operations, extending the generator and runner of the existing implementation to support regular expressions was rather straightforward.

One of the challenging parts was parsing and validating the returned models of type regular expressions, as the java API had some parsing bugs. This issue was reported by us and fixed by the developers of Z3 (issue [5]).

5.5 Limitation of the implementation

Due to our technique being exhaustive and having to build an automaton for each regular expression, to perform equality checks, our implementation often runs out of memory. As a consequence we needed to restrict the number of the predefined constants, to some small number. Thus also not allowing us to build a complex random regular expression pool.

There were attempts of lifting the constraint of being exhaustive in the transformations, allowing us to use a complex randomized regular expression pool and perform multiple rounds of a transformation. However, at the time of writing, this work is not complete for all transformations. Finalizing this task is left as future work.

Chapter 6

Evaluation

In this chapter we present and discuss the results for our considered solvers. We show that our technique was able to detect soundness errors, completeness issues as well as performance issues. Furthermore we will discuss the limitations of the technique for regular expressions.

6.1 Experimental setup

The solvers were run with a fixed random seed (42). We set a timeout of 15 seconds for each test. Furthermore options produce-models and produce-unsat-cores were enabled for the solvers. For CVC5 we used the option "strings-exp" to enable non-primitive string operations. The experiments were performed on an Intel Core i7-7700K CPU @ 4.20GHz with 16 GB ram and 400 GB additional swap memory.

6.1.1 Not supported operations

We did not consider all operations from Table 3.2. At the point of writing the *re_replace*, *re_replace_all* were not supported by Z3. Furthermore the power operation was added to the Java API (issue [8]) after having conducted the experiments and thus also not included in our experiments.

The loop operation does not allow its integer arguments to be variables. As a consequence we left the integers used in the loop operation constant for all transformations.

6.2 Non exhaustiveness

In principal our method is designed to be run exhaustively. Nevertheless due to time and memory issues, we decided to be non exhaustive for the

Table 6.1: Experimental results for Z3_seq 4.8.14

Theory	Expected	Transformation	#Tests	IM	IC	S	U	K	T	E
Regex	sat	operation	11	0	0	11	0	0	0	0
Regex	sat	constant assignment	814	0	0	618	11	132	53	0
Regex	sat	term synthesis (2.3.2)	2689	0	0	523	0	1543	544	79
Regex	unsat	const (2.5.1)	993	0	0	0	993	0	0	0
Regex	unsat	constComplexResult (2.5.1)	1534	0	0	0	1533	1	0	0
Regex	unsat	larger unsat core (2.4.1, 2.4.2)	81	0	0	0	0	1	80	0
Combined	sat	term synthesis	278	0	0	15	0	136	127	0
Combined	unsat	constComplexResult	865	0	0	0	857	0	8	0

IM=incorrect model; **IC**=incorrect unsat core; **S**=sat; **U**=unsat; **K**=Unknown;
T=timeout; **E**=error

transformation presented in Section 2.5.2 where we limited the amount of complex results to some configurable value (10 in our experiments).

The two transformations for combined theories were also not run exhaustively. We limited the generated formulas of each operation to some configurable value (20 in our experiments) to avoid memory issues.

6.3 Testing latest versions of SMT solvers

In our evaluation we consider the latest solvers of Z3_seq [13] (version 4.8.14), CVC5 [1] (version 0.0.7) and Z3str3 (version 4.8.14) [13].

6.3.1 Z3_seq 4.8.14

In this section we discuss the most important results for the latest version of Z3_seq. These are summarized in Table 6.1. In this table we specify the theory (column 1) which can either be Regex or Combined (combining regex and arrays/bitvectors), the expected result (column 2) which is either sat or unsat, the transformation (column 3),as described in Chapters 2 and 3, used to generate the formulas and the total number of tests generated (column 4). The remaining columns specify the actual result returned by the solvers. Solvers can return an incorrect model (correctly return sat, but the produced model is not valid with respect to our executable semantics), return an incorrect unsat core (correctly return unsat, but an incorrect unsat core), return sat, return unsat, return unknown, they can timeout or return an error (solver returned an error message).

Constant assignment transformation

We discovered a total of 11 unsoundly returned unsat cases (marked with red in Table 6.1). One of the unsound test cases was described in Figure 2.3 for the range operation involving only constants. The other 10 unsound cases involved regex variables and contained either the difference, intersection,

$$intersection(tmp\_regex0, re\_allchar) = intersection(a, re\_allchar)$$

Figure 6.1: Sat formula obtained by constant assignment transformations for which Z3_seq 4.8.14 unsoundly returned unsat, where tmp_regex0 has type Regular expression.

kleene_closure or option operation. One other instance of a sat formula for which the solver unsoundly returned unsat is shown in Figure 6.1.

As we used both constants a and b in our regex pool, we do not expect all 11 unsat cases to be caused by different bugs. For instance we also obtained unsat for using b instead of a in Figure 6.1.

Constant assignment transformation was very effective and we managed to achieve our goal of exposing soundness bugs.

Larger unsat core

Increasing the unsat core of a formula by using Constant replacement transformation or Variable replacement transformation (transformation described in Section 2.4) was very effective in exposing performance issues. Out of 81 test cases Z3_seq timed out 80 times. We experimented with increasing the timeout to one minute, Z3_seq 4.8.14 still timed out. Although it was not tested for even larger timeout values, we assess a change of outcome to be rather unlikely when increasing the timeout value further.

Unsat formulas with only constants

Z3_seq performed very well for formulas generated by the transformations described in Sections 2.5.1 and 2.5.2. Z3_seq correctly returned unsat for 2526 out of 2527 tests. For one test case the solver returned unknown, thus this test case exposed a completeness issue.

Combined

Combined term synthesis exposed completeness issues as well as performance issues. Constant and complex result for combined formulas exposed further performance issues. Note that Z3_seq performs significantly better if formulas only contain constants.

6.3.2 CVC5 0.0.7

Testing the latest version of CVC5, we obtained many errors, as CVC5 does not support regular expression equality in general (CVC5 can only reason on a string representation level) nor does it support regex variables. Our technique did not expose any soundness issues, nor did it expose completeness or performance issues in CVC5. The results are presented in Table 6.2.

Table 6.2: Experimental results for CVC5 0.0.7

Theory	Expected	Transformation	#Tests	IM	IC	S	U	K	T	E
Regex	sat	operation	11	0	0	0	0	0	0	11
Regex	sat	constant assignment	814	0	0	210	0	0	0	604
Regex	sat	term synthesis (2.3.2)	2689	0	0	53	0	0	0	2636
Regex	unsat	const (2.5.1)	993	0	0	0	35	0	0	958
Regex	unsat	constComplexResult (2.5.1)	1534	0	0	0	53	0	0	1481
Regex	unsat	larger unsat core (2.4.1, 2.4.2)	81	0	0	0	0	0	0	81
Combined	sat	term synthesis	278	0	0	0	0	0	0	278
Combined	unsat	constComplexResult	865	0	0	0	24	0	0	841

IM=incorrect model; **IC**=incorrect unsat core; **S**=sat; **U**=unsat; **K**=Unknown; **T**=timeout; **E**=error

$$kleene\_closure(tmp\_regex0) = kleene\_closure(re\_allchar)$$

Figure 6.2: Formula for which Z3str3 4.8.14 returns an incorrect model: $tmp\_regex0 = string\_to\_regex(''!0!'')$ and Z3_seq 4.8.9 returns unsat.

Table 6.3: Experimental results for Z3str3 4.8.14

Theory	Expected	Transformation	#Tests	IM	IC	S	U	K	T	E
Regex	sat	operation	11	0	0	11	0	0	0	0
Regex	sat	constant assignment	814	190	0	615	1	8	0	0
Regex	sat	term synthesis with regex var.	2689	1797	0	556	0	320	16	0
Regex	unsat	const (2.5.1)	993	0	0	613	380	0	0	0
Regex	unsat	constComplexResult (2.5.1)	1534	0	0	986	548	0	0	0
Regex	unsat	larger unsat core (2.4.1, 2.4.2)	81	0	0	13	68	0	0	0
Combined	sat	term synthesis	278	184	0	51	0	33	10	0
Combined	unsat	constComplexResult	865	0	0	162	703	0	0	0

IM=incorrect model; **IC**=incorrect unsat core; **S**=sat; **U**=unsat; **K**=Unknown; **T**=timeout; **E**=error

6.3.3 Z3str3 4.8.14

The results for Z3str3 4.8.14 are summarized in Table 6.3. Z3str3 4.8.14 often returned incorrect models for formulas using regex variables such as Figure 6.2. This formula was reported in the Z3-issue tracker and remained an open issue [16] at the time of writing. Furthermore for unsat formulas with only constants and complex result (described in Section 2.5.2) 986 tests out of 1534 were unsoundly returned sat. Overall I believe Z3str3 4.8.14 should not be used to solve formulas with regex variables and formulas solving regex equality as the results show.

6.4 Known issues

In this section we compare our generated tests evaluated with Z3_seq 4.8.9 (described in Table 6.4) with known reported issues. For Z3_seq we looked at tests in the Z3-Issue tracker [14] starting from Sep 11, 2020 where the new regular expression solver was integrated until October 17, 2021. We only looked at tests containing regular expression operations. The considered issues are listed in Appendix A. Out of 47 tests we were able to reproduce

Table 6.4: Experimental results for Z3_seq 4.8.9

Theory	Expected	Transformation	#Tests	IM	IC	S	U	K	T	E
Regex	sat	operation	11	0	0	11	0	0	0	0
Regex	sat	constant assignment	814	44	0	566	5	49	150	0
Regex	sat	term synthesis (2.3.2)	2689	0	0	516	0	1188	960	25
Regex	unsat	const (2.5.1)	993	0	0	0	969	0	24	0
Regex	unsat	constComplexResult (2.5.1)	1534	0	0	0	1530	0	4	0
Regex	unsat	larger unsat core (2.4.1, 2.4.2)	81	0	0	0	0	0	81	0

IM=incorrect model; **IC**=incorrect unsat core; **S**=sat; **U**=unsat; **K**=Unknown;
T=timeout; **E**=error

0 tests with our technique. To give an explanation, we identify common patterns in the reported issues in the following section.

6.4.1 Common patterns in the reported issues

Regular expression Variables Out of 47 tests only one reported issue [4] contained regular expression variables. String variables on the other hand were present in almost all tests that caused bugs. Only few tests consisted of constants only.

String_in_lang operation Most issues were caused using the string_in_lang operation as the most outer operation. Most reported issues containing the string_in_lang operation compared CVC5's [1] result to the result of Z3, which is an alternative approach for testing the solvers [21]. Note that regex equality is not supported by CVC5 and one cannot do differential testing for Z3 using CVC5 as a reference for formulas solving regular expression equality.

Multiple clauses for satisfiable formulas. Some tests contained multiple clauses for satisfiable formulas.

Nested operations. Strikingly many reported issues contained multiple nested operations.

If statement operation. Many issues contained if-statements in satisfiable formulas. Note that our generated satisfiable tests do not contain the if-statement.

6.4.2 Comparing our identified issues to the reported issues

The tests that were unsoundly returned unsat for our generated formulas by Z3_seq 4.8.9 were formulas, solving regex equality with regex variables such as Figure 6.2 or contained only constants. Whereas the reported issues mostly contain the string_in_lang operation with multiple nested operations, containing String variables and no regex variables.

Table 6.5: Z3_seq 4.8.14 Term synthesis comparison

Theory	Expected	Transformation	#Tests	IM	IC	S	U	K	T	E
Regex	sat	term synthesis with regex var.	2689	0	0	523	0	1543	544	79
Regex	sat	term synthesis without regex var.	1103	1	0	1096	0	4	2	0

IM=incorrect model; **IC**=incorrect unsat core; **S**=sat; **U**=unsat; **K**=Unknown;
T=timeout; **E**=error

Table 6.6: Z3_seq 4.8.9 Term synthesis comparison

Theory	Expected	Transformation	#Tests	IM	IC	S	U	K	T	E
Regex	sat	term synthesis with regex var.	2689	0	0	516	0	1188	960	25
Regex	sat	term synthesis without regex var.	1103	0	0	783	0	0	320	0

IM=incorrect model; **IC**=incorrect unsat core; **S**=sat; **U**=unsat; **K**=Unknown;
T=timeout; **E**=error

6.5 Impact of regular expression variables for Z3_seq

For term synthesis with regex variables (presented in Section 2.3.2) out of 2689 tests Z3_seq 4.8.14 returned unknown or timed out for 2087 (described in Table 6.6). If we follow the approach from Section 2.3.3 for term synthesis only 6 out of 1103 tests result in unknown or timeout. This shows us, that formulas with regex variables are very hard to solve.

Term synthesis without regex variables was able to detect a test case for which Z3_seq returned an incorrect model shown in Figure 2.7.

Term synthesis without regex variables in Z3_seq 4.8.9 (shown in Table 2.7) resulted in a total of 320 timeouts or unknowns, whereas its latest version only results in a total of 6 timeouts or unknowns, which is a great improvement of the latest version.

6.6 Limitation of the technique

As shown in Section 6.3.2, CVC5 does not support regex variables and the support for regex equality is very limited. Yet our technique generates only few test cases that are supported by CVC5 in general (string_in_lang operation with no regex variables).

In Section 6.4 we have shown that many reported issues in [14] contain multiple nested operations. Due to memory issues as shown in Section 5.5 our technique is not able to generate formulas containing multiple nested operations.

Chapter 7

Related work

In this chapter we discuss alternative approaches for testing SMT-solvers.

Differential testing

Differential testing [21] is a widely used approach for testing SMT-solvers. One tests a formula on two different solvers and compares the result produced by the solvers. If the results are different, it points to a bug in one of the solvers. Most reports in the Z3-issue tracker [14] for formulas containing regular expression, refer to a different result obtained in CVC5 and thus applying differential testing.

One cannot use differential testing (with solvers CVC5 and Z3) for formulas containing regular expressions variables and containing regex equality, as these functionalities are not supported by CVC5 yet. Whereas our technique can determine bugs in formulas containing regex variables and regex equality in Z3.

Semantic Fusion

Semantic fusion [20] is an effective approach to generate SMT formulas. The key concept is to fuse two tests (both sat or both unsat) into an equisatisfiable formula that can then be used to test the solvers. One considered theory in [20] is regex. They present a formula with regex, exposing a soundness bug for Z3 which is not obtainable by our technique.

Chapter 8

Conclusions

In this thesis we have extended the transformations proposed in [19] to also support regular expressions. Furthermore we have explored other transformation techniques only involving constants [18] and extended them to support regular expressions. We also considered the generation of formulas combining arrays, bitvectors and regex. Our evaluation shows that all transformations are able to expose performance, as well as incompleteness issues for state-of-the art SMT solvers. Some of them can even detect soundness issues in the solvers.

Additionally we have shown that state-of-the art solvers do not have good support for regex variables and regex equality. CVC5 does not support regex variables/regex equality, Z3str3 often returns incorrect models. The only solver I consider appropriate for solving formulas with regex equality is Z3_seq. Our evaluation shows that Z3_seq reliably answers queries solving regex equality containing only constants. However our evaluation also shows, that Z3_seq often returns unknown or timeout for formulas containing regex variables and thus also not offering good support for regex variables. Overall I believe, that the evaluation shows, that formulas with regex variables and regex equality are very hard to solve for SMT solvers in general.

8.1 Future work

In this section we present future work, extending our presented approach.

Replace operation. In Section 6.1.1 we have shown that not all operations are supported by Z3 yet. The replace operations involving regex are likely to be added to Z3 in the future. Adding these replace operations to our implementation would not require much effort, as the framework for regex is already implemented.

Additional transformations. We adapted variable/constant replacement transformation for regular expressions but we did not consider this technique for combination of arrays/bitvectors and regex. This would require new rewriting rules (similar to Table 2.6) involving multiple theories.

Non exhaustive methods. Our proposed technique is mainly designed to be exhaustive. Due to memory issues of the implementation, we cannot do multiple rounds of, for instance, term synthesis. Lifting the constraint of being exhaustive would enable us to generate more complex formulas, at the cost of possibly missing interesting test cases for the solvers. As many bug reports in the Z3-issue tracker contain multiple nested operations, this might be an effective approach for detecting issues in the solvers.

Sat formulas with quantifiers. Our formulas do not contain universal quantifiers. In universally quantified sat formulas, the model for a free variable occurring in universally quantified clauses has to hold for all possible values for the quantified variables, which is complicated. We were not able to find a method of adapting our transformations for universally quantified sat formulas and are rather convinced that a new theoretical approach is needed.

Appendix A

List of known issues in the Z3-Issue tracker

- https://github.com/Z3Prover/z3/issues/5140 (25 tests containing regex)
- https://github.com/Z3Prover/z3/issues/5298 (1 test containing regex)
- https://github.com/Z3Prover/z3/issues/5390 (1 test containing regex)
- https://github.com/Z3Prover/z3/issues/5467 (7 tests containing regex)
- https://github.com/Z3Prover/z3/issues/5591 (8 tests containing regex)
- https://github.com/Z3Prover/z3/issues/5603 (5 tests containing regex, within considered time span)

Bibliography

[1] CVC5. https://cvc5.github.io/.

[2] Dk brics automaton. https://www.brics.dk/automaton/.

[3] Dk brics RegExp. https://www.brics.dk/automaton/doc/dk/brics/automaton/RegExp.html.

[4] Issue 5467. https://github.com/Z3Prover/z3/issues/5467.

[5] Java API issue. https://github.com/Z3Prover/z3/issues/5855.

[6] Java JDK 11.0.14. https://www.oracle.com/java/technologies/javase/11-0-14-relnotes.html.

[7] Java regex. https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html.

[8] RePower request issue. https://github.com/Z3Prover/z3/issues/5872.

[9] SMT-LIB Arrays. https://smtlib.cs.uiowa.edu/theories-ArraysEx.shtml.

[10] SMT-LIB Bitvectors. https://smtlib.cs.uiowa.edu/theories-FixedSizeBitVectors.shtml.

[11] SMT-LIB Strings and Regular expressions. http://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml.

[12] Stringsolvers implementation. https://github.com/alebugariu/StringSolversTests.

[13] Z3. https://github.com/Z3Prover/z3.

[14] Z3-Issuetracker. https://github.com/Z3Prover/z3/issues.

[15] Z3 Java API documentation. https://z3prover.github.io/api/html/namespacecom_1_1microsoft_1_1z3.html.

[16] Z3str3 issue. https://github.com/Z3Prover/z3/issues/5873.

[17] Z3str3RE. https://z3string.github.io/z3str3RE/readme.html.

[18] Olivier Becker. Automatically Testing SMT Solvers. https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Olivier_Becker_BA_Report.pdf, 2021.

[19] Alexandra Bugariu and Peter Müller. Automatically testing string solvers. https://doi.org/10.1145/3377811.3380398, 2020.

[20] Chengyu Zhang Dominik Winterer and Zhendong Su. *Validating SMT Solvers via Semantic Fusion*. 2020.

[21] W. M. McKeeman. *Differential Testing for Software*. 1998.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

AUTOMATICALLY TESTING SOLVERS FOR STRING AND REGULAR EXPRESSIONS CONSTRAINTS

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
KÜHNE	SEBASTIAN

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Adligenswil, 01.04.2022	S. Kühne

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.