Automatically Testing Solvers for String and Regular Expressions Constraints

Sebastian Kühne
supervised by Prof. Dr. Peter Müller, Alexandra Bugariu

October 2021

1 Problem Description

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. For instance the regular expression in

"`(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9]).{8,32}`"

Figure 1: String Regular expression for a password format [1]

Figure 1 matches all Strings with minimum eight characters and maximum 32 characters (purple), at least one lower-case letter (blue), at least one capital letter (red) and at least one number (green).

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

Figure 2: Satisfiable SMT formula in mathematical notation where s has type String.

Satisfiability modulo theories (SMT) formulas are expressed in first-order-logic. They can support typical theories such as Arrays and Integers but in our thesis we will focus on SMT formula with *strings and regular expressions*. For instance Figure 2 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.

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 2 we can assign "ad" to s. Because "ad" is in L this function 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.

$$string\_in\_lang(s, \emptyset) = true$$

Figure 3: Unsatisfiable SMT formula in mathematical notation where s has type String and $\emptyset$ is a regular expression describing the empty language.

An SMT formula is unsatisfiable if there does not exist a model. The SMT formula in Figure 3 evaluates to false for all possible values of the free variable s as the regular expression $\emptyset$, 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*. Unfortunately they can also time out (they cannot solve the query within a given time span). 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.

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

Figure 4: Formula that exposes a soundness bug in Z3 (version from April 2020)

The formula from Figure 4 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 [3] 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 [4] (version from April 2020) incorrectly returned sat which is a soundness issue.

Our goal is to detect these issues by extending the testing technique proposed in [2] to also support regular expressions.

Table 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	$replace(s, re, t)$	$replace\_all(s, re, t)$	
Boolean	$string\_in\_lang(s, re)$		

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

2 Approach

We will start from the technique described in [2] which automatically generates SMT formulas from the string theory and extend it to support regular expressions, as well as combinations between string operations and regular expressions. We plan to expand the technique from [2] to support operations from Table 1 whose semantics are described in detail in *SMT-LIB* [3].

$$concat(re_1, re_2) = res$$

Figure 5: Simple satisfiable formula generated in step 1. All variables have type Regular expression.

2.1 Creating satisfiable formulas

In step 1 we take formulas involving an operation with regular expressions described in Table 1 with unconstrained parameters and an unconstrained result. Thus those formulas are trivially satisfiable. An example for a formula generated in step 1 is shown in Figure 5.

In step 2 we want to extend the satisfiability preserving transformations proposed in [2] and use them to obtain more complex satisfiable formulas.

In the following, we illustrate one of these transformations, i.e. the *constant assignment transformation for satisfiable formulas.*

2.1.1 Regular expression constant assignment transformation

Constant assignment transformation relies on concrete execution. Thus we need to extend the implementation of [7] to also support regular expression operations. Therefore we need to implement all operations from Table 1 according to the *reference semantics* of [3] to have an executable version of our operations.

In order to apply constant assignment transformation [2], we need to setup a pool of predefined constant regular expressions as [2] did for Integers and Strings. We propose a simple technique for creating a pool of regular expressions, described below.

2.1.2 Building a constant regular expression pool suitable for constant assignment transformation:

The following procedure relies again on concrete execution and generates arbitrarily many constant regular expressions:

1. Build a string and integer pool, for instance as proposed in [2]: $integer\_pool = \{-1, 0, 1\}$ and *string_pool* consists of the empty string, strings of length 1 and strings containing quotes.
2. Initialize an empty *regular_expression_pool*.
3. Evaluate $all()$ and add the result, which is the regular expression describing all strings, to *regular_expression_pool*. Proceed analogously for the operations $all\_char()$ and $none()$.
4. Add evaluated $string\_to\_regex(s_1)$ and evaluated $range(s_1, s_2)$ (regular expression describing all strings "between" s_1 and s_2 if length of $s_1, s_2 = 1$ and the empty language otherwise) to *regular_expression_pool* for some $s_1, s_2 \in string\_pool$.
5. Evaluate operations from Table 1 on constant arguments of type Regular expressions from *regular_expression_pool* and add the result to the pool. For instance we evaluate $concat(re_1, re_2)$ where re_1, re_2 are some constant regular expressions of the pool and add the result to the pool.
6. Repeat step 5 until a certain size of the pool is reached.

Having built a *regular_expression_pool*, we can apply regular expression constant assignment transformation [2] to simple SMT formulas containing regular expressions. We illustrate the constant assignment technique on Figure 5. For illustration purposes the *regular_expression_pool* contains, among other regular expressions, the regular expressions `"[a-c]"` and `"[d-f]"`. We concretely execute $concat(re_1, re_2)$ for all $re_1, re_2 \in regular\_expression\_pool$. One concrete execution is: $concat($`"[a-c]"`$,$`"[d-f]"`$) =$ `"[a-c][d-f]"`. We can now substitute some of the free variables of Figure 5 with values from the concrete execution which then yields the formula of Figure 6.

$$concat(re_1, \texttt{"[d-f]"}) = \texttt{"[a-c][d-f]"}$$

Figure 6: More complex satisfiable formula obtained by applying the constant assignment transformation on the formula from Figure 5 where re_1 has type Regular expression.

2.2 Creating unsatisfiable formulas

Creating unsatisfiable formulas requires a different approach. We cannot rely on concrete execution as a formula is unsatisfiable if every assignment of the free variables is unsatisfiable. Unfortunately if our free variables are regular expressions or strings we cannot concretely execute our formula as there are infinitely many assignments of string free variables and regular expression free variables. Thus we can not use concrete execution to prove that a formula with free variables is unsatisfiable.

[2] proposed a solution to this problem. They consider an input formula $F = A \wedge \neg B$ where A and B are equivalent. F is unsatisfiable by construction because A is equivalent to B and $A \wedge \neg A$ is trivially unsatisfiable. They then transform F into more complex formulas.

We extend the unsatisfiability preserving transformations proposed in [2] to also support regular expressions and use them to obtain more complex unsatisfiable formulas. In the following, we illustrate one of these transformations, i.e. the *constant assignment transformation for unsatisfiable formulas.*

$$\mathit{option}(re) = res \iff res = \mathit{union}(re, \texttt{""})$$

Figure 7: Equivalent formulas for a non primitive regular expression operation where re, res have type Regular expression and `""` is the regular expression, describing the language only containing the empty String.

2.2.1 Regular expression constant assignment transformation

In the first step we consider simple formulas $F(x, y) = \neg A(x) \wedge B(c, y)$ where A and B are equivalent and c is a constant. These simple formulas are built from equivalent formulas for a non primitive regular expression operation. An example for such equivalent formulas is shown in Figure 7. The $\mathit{option}(re)$ operation describes the language of re united with `""`. Thus the formulas from Figure 7 are equivalent. We can now build the simple formula from Figure 8 out of these equivalent formulas.

$$\neg(\mathit{option}(re) = res) \wedge res = \mathit{union}(re, \texttt{""})$$

Figure 8: Simple unsatisfiable formula built from the equivalent formulas from Figure 7 where re, res have type Regular expression

In the second step we transform the simple formula into a more complex unsatisfiable formula. We substitute the constant c by a new free variable z_{fresh} and add an additional clause $C(c, z_{fresh})$ which implies that $z_{fresh} = c$. For the formula from Figure 8 we can take for instance the clause $C(\texttt{""}, z_{fresh})$ from Figure 9 which then yields the new more complex unsatisfiable formula shown in Figure 10.

$$C(\texttt{""}, z_{fresh}) \coloneqq concat(\emptyset, z_{fresh}) = \texttt{""}$$

Figure 9: Clause $C(\texttt{""}, z_{fresh})$, where z_{fresh} has type Regular expression, implies that $z_{fresh} = \texttt{""}$.

$$\neg(option(re) = res) \wedge res = union(re, z_{fresh}) \wedge concat(\emptyset, z_{fresh}) = \texttt{""}$$

Figure 10: Unsatisfiable formula obtained by applying regular expression constant assignment transformation where re, res, z_{fresh} have type Regular expression.

3 Main Challenges

3.1 Implementation of regular expression operations

We will need to carefully implement all operations of Table 1, according to the reference semantics of [3], because we rely on concrete execution when building satisfiable formulas as described in Section 2.1. Even a slight mistake in our implementation could lead to an unsatisfiable formula opposed to a desired satisfiable formula.

3.2 Creating unsatisfiable formulas with constants

Some unsatisfiable formulas cannot be generated using the approach from Section 2.2. For example we cannot obtain the formula of Figure 4 by transforming an initial unsatisfiable formula $F = A \wedge \neg B$ where A, B are equivalent. However this formula does not contain any free variables. We propose a technique to automatically generate these type of formulas, similar to a technique which was first mentioned in [6], that relies on concrete execution for building unsatisfiable formulas.

We concretely execute $string\_in\_lang(s, re)$ with values from our string and regular expression pool. We can then build a more complex formula by taking the left-hand side equal to $string\_in\_lang(s, re)$ with s, re from the pools and the right-hand side equal to the negation of their evaluated boolean result. If $s = ""$ is in our string pool and $loop(189, 0, string\_to\_regex(""))$ in our regular expression pool one of the concrete execution yields:
$string\_in\_lang("", loop(189, 0, string\_to\_regex(""))) = false$. If we now negate the boolean result we get the unsatisfiable formula:
$string\_in\_lang("", loop(189, 0, string\_to\_regex(""))) = true$ which can be trivially simplified to the formula from Figure 4.

3.3 Adapting the string pool and the integer pool

[2] proposes the string pool to consist of the empty string, strings of length 1 and strings containing quotes. In order to find an assignment of the free variables for the formula of Figure 2 such that the formula is satisfiable, we need to assign a

string of length 2 for instance "ad" to s. Thus we need to rethink the predefined set of Strings because we introduced regular expressions.

Analogously, we need to adapt the integer pool as well because we introduced the *loop* operation from Table 1 which involves integers.

4 Core Goals

In this project, we plan to address the following core goals:

- Create a pool of representative regular expressions, following the approach described in Section 2.1.2.
- Design a theoretical solution that extends the technique used in [2] to automatically generate increasingly complex satisfiable formulas with regular expressions. For each formula, we will also generate a possible model. We will follow the approach from Section 2.1.
- Design a theoretical solution that extends the technique used in [2] to automatically generate increasingly complex unsatisfiable formulas with regular expressions, for which the minimal unsat core is known by construction. We will follow the approach from Section 2.2.
- Design a theoretical solution for generating unsatisfiable formulas with constants, generalizing the ideas proposed in [6]. We will follow the approach from Section 3.2.
- Extend the implementation [7] to also include regular expressions.
- Evaluate our solution on state-of-the-art SMT solvers such as Z3 [4] and CVC4 [5]. We will use previous versions of those solvers to compare our found bugs with previously reported bugs and the newest version to possibly detect new bugs.

5 Extension Goals

In this project, the following extension goals might also be addressed:

- Test our solution on the latest solvers supporting regular expression constraints such as dZ3 [8] and Z3str3RE [9].
- Automatically identify common patterns in the failed tests and express them as regular expressions. This would avoid duplicated bug reports and will facilitate error localization.
- Generate satisfiable formulas with strings and regular expressions which contain universal quantifiers.

6 Schedule

Week	**Date**	**Plan**
Week 1	18.10.2021-24.10.2021	Initial presentation
Week 2	25.10.2021-31.10.2021	Create a pool of representative regular expressions
Week 3	01.11.2021-07.11.2021	Theoretical solution for satisfiable formulas
Week 4	08.11.2021-14.11.2021	Theoretical solution for satisfiable formulas
Week 5	15.11.2021-21.11.2021	Implementation for satisfiable formulas
Week 6	22.11.2021-28.11.2021	Implementation for satisfiable formulas
Week 7	29.11.2021-05.12.2021	Theoretical solution for unsatisfiable formulas (with free variables, as in Section 2.2)
Week 8	06.12.2021-12.12.2021	Theoretical solution for unsatisfiable formulas (with free variables, as in Section 2.2)
Week 9	13.12.2021-19.12.2021	Theoretical solution for unsatisfiable formulas (with constants, as in Section 3.2)
Week 10	20.12.2021-26.12.2021	Implementation for unsatisfiable formula
Week 11	27.12.2021-02.01.2022	Implementation for unsatisfiable formulas
Week 12	03.01.2022-09.01.2022	Evaluate solution
Week 13	10.01.2022-16.01.2022	Evaluate solution
Week 14	17.01.2022-23.01.2022	Extension goals
Week 15	24.01.2022-30.01.2022	Extension goals
Week 16	31.01.2022-06.02.2022	Write thesis
Week 17	07.02.2022-13.02.2022	Write thesis
Week 18	14.02.2022-20.02.2022	Write thesis
Week 19	21.02.2022-27.02.2022	Final presentation

References

[1] Password regular expression,
`https://www.ocpsoft.org/tutorials/regular-expressions/password-regular-expression/`

[2] Alexandra Bugariu and Peter Müller (2020) *Automatically testing string solvers*, In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20). Association for Computing Machinery, New York, NY, USA, 1459–1470. DOI:`https://doi.org/10.1145/3377811.3380398`

[3] SMT-LIB, *The Satisfiability Modulo Theories Library.*
`http://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml`

[4] Z3,
`https://github.com/Z3Prover/z3`

[5] CVC4,
`https://cvc4.github.io/`

[6] Project description of Automatically Testing SMT Solvers by Olivier Becker,
`https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Olivier_Becker_BA_Description.pdf`

[7] StringSolversTests by Alexandra Bugariu,
`https://github.com/alebugariu/StringSolversTests`

[8] dZ3 solver,
`https://github.com/cdstanford/dz3-artifact`

[9] Z3str3RE,
`https://z3string.github.io/z3str3RE/readme.html`