Search-based software test data generation for string constraints

Ohjun Kwon
ohjun@kaist.ac.kr
KAIST
Daejeon, South Korea

Jonguk Jeon
ukj0806@kaist.ac.kr
KAIST
Daejeon, South Korea

Seungjun Chung
s.j.chung@kaist.ac.kr
KAIST
Daejeon, South Korea

Abstract

Many automated tests for programs can be effectively reduced to a constraint solving phase. Recently, various solvers for string constraint have come up yet to be applied only to fix sized string inputs. To address this problem, this paper proposes a novel approach of utilizing alternative variable method. Alternative Variable Method(AVM) is one of heuristic local search algorithms. AVM approach to automated software test data generation successfully covers maximum branch on string constraints - string equality and string ordering. This approach shows increase in speed and performance when dealing with nested constraints compared to random test data generation.
Software and data are available online at: https://github.com/kojandy/AVM

[This paper is for a CS453 course project.]

1 INTRODUCTION

Many automated tests for programs can be effectively reduced to a constraint solving phase. This separation of concerns often leads to development of effective tools. Recently, various constraint solvers for string constraint have come up, which makes their efficacy compelling[4]. These off-the-shelf solvers focus on SMT(satisfiability modulo theory) on string[1][5] or constraints over fixed size string[3]. However, there are few or less ones that deal with undetermined size string. String constraints are constraints that consist of only string variables and constants. In this research, we considered particular constraints under following conditions:

- Operator for string equality: ==, !=
- Operator for string ordering: <, <=, >, >=
- LHS/RHS: string constant, string variable, slice of them.
- Slice: substring, charAt with constant index.
- Constraint consists of a operator of the above, a left hand side and a right hand side of the above.

For instance, such constraint is possibly solved by ours:

```
string a, b;
if(a[2:4] == b[0:2])
```

Solving the above successfully yield the result:

```
'a' : 'tsjGR', 'b' : 'jGCD'
```

Ways to handle string constraint is very much different with dealing with integer constraint. Particularly in search-based field of research, it is inevitable to calculate branch distance from a certain input. There are two big issues here.

First, in the case of manipulating integer constraints there are only two directions, either getting closer or getting far which corresponds to increment or decrement. However, since strings are manipulated by character by character, there are far more ways to manipulate a single string.

Second, relational operators gain totally different definition in integer and string. For instance in string, relational operator ">" compares from the head regardless of its length, so that "z" is estimated greater than "aaaaaa." On the other hand, integers get compared with their length first and then compared from the head.

Such problems arise in the string constraint which makes it much more difficult to resolve compared to integer constraint. Surely there are related works conducted. However, most of them focus on resolving SMT(satisfiability modulo theory) on string. Some of the remaining do deal with actually solving a given string constraint, but ended up only handling fix sized string inputs. In this paper, we propose a novel approach of applying AVM to generate test data for string constraints.

Alternative Variable Method(AVM) is one of heuristic local search algorithms that optimizes a vector with multiple variables to some object function. There are two types of move in AVM, exploratory moves and pattern moves. First, exploratory moves choose which direction among increase and decrease of a selected variable improves function value. Second, pattern moves change the value of the selected variable to chosen direction gradually until making better function value. The optimal vector can be found after repeating the two moves for a variable and changing variable.

String can be considered as a vector with define corresponding integer to each characters. We defined fitness functions of string constraints for object function of AVM. We made automated software test data generation using AVM.

2 RELATED WORK

There are several off-the-shelf string constraint solvers. Some of them do really solve string constraint[5], whereas the others focus on resolving SMT(satisfiability modulo theory) on string[1][5]. Hampi[3] is a solver for string constraints over fixed-size string variables. Hampi constraints express membership in regular languages and fixed-size context-free languages. The main difference in our work is that Hampi needs a constant size information in order to resolve the constraints. We narrowed down the target domain to cover particular operators within it and successfully resolved constraints without any information of size of string input.

To cover one more major difference, existing works give definite answer to satisfiability problem using their own algorithm. If some constraints have no matching string input, they yield such result. On the contrary, ours also make similar decision but in different way which is a limited number of function call. Ours has an internal structure that counts number of function call, so that if some constraints cannot be resolved within certain number of loop it answers "-" meaning unsatisfiable constraint. However arguing the genuine of the answer is out of scope of this research.

There is an existing approach on SBST for string using global search algorithm particularly genetic algorithm[2]. Genetic algorithm is motivated by Darwin's the survival of the fittest genetic theory.

3 METHOD

3.1 Branch distance

3.1.1 *String equality*

We define branch distance of equality of two string with the same length as sum of each character's distance. Character distance of two characters is an integer meaning how far away two characters are. We match each character to integer in order such as 'a' to 1, 'b' to 2 and 'z' to 26. Character distance of two characters is defined as an absolute value of difference of corresponding two integers.

To get branch distance of strings with different length, simply add a product of length difference and number of characters. However, our algorithm only need branch distance with same length strings as we define above.

An example is as follows. $CD(x, y)$ denotes character distance of two character x, y and $BD(=, X, Y)$ denotes branch distance of two string X, Y.

$$CD(`a\text{'}, `c\text{'}) = 2$$
$$CD(`e\text{'}, `b\text{'}) = 3$$
$$\begin{aligned} &BD(=, “abc”, “xyz”) \\ &\quad = CD(`a\text{'}, `x\text{'}) + CD(`b\text{'}, `y\text{'}) + CD(`c\text{'}, `z\text{'}) \\ &\quad = 69 \end{aligned}$$

3.1.2 *String inequality*

We define branch distance of "!=" case of strings as negative value of sum of each characterâĂŹs distance. An example is as follows. $BD(\neq, X, Y)$ denotes branch distance of two string X, Y.

$$\begin{aligned} &BD(\neq, “abc”, “xyz”) \\ &\quad = -(CD(`a\text{'}, `x\text{'}) + CD(`b\text{'}, `y\text{'}) + CD(`c\text{'}, `z\text{'})) \\ &\quad = -69 \end{aligned}$$

3.1.3 *String ordering*

We define $OV(X)$, ordinal value of string X as following formula while 'n' denotes length of string X and '$numCharacter$' denotes number of characters.

$$OV(X) = \sum_{i=0}^{n-1} numCharacter^{(n-i-1)} \times (\mathrm{ord}(X[i]) - 96)$$

We define a branch distance of two string X, Y as below.

$$BD(>, X, Y) = OV(Y) - OV(X) + 1$$
$$BD(\geq, X, Y) = OV(Y) - OV(X)$$
$$BD(<, X, Y) = OV(X) - OV(Y) + 1$$
$$BD(\leq, X, Y) = OV(X) - OV(Y)$$

3.2 Calculating minimum length

Calculating minimum length is an important process in that it significantly decreases overhead. Without it, an appropriate string input should have been searched from of length zero. However, it successfully provides a minimum length of each string variables to start with. Need for this notably arises when dealing with slices. For instance, consider the following:

```
string a;
if(a[3:7] == "abcd")
```

All know that string variable a should have length of minimum seven. The minimum length calculating function browses through target string constraints and give minimum length for each of all string input variables to be used. Type of the function output is *dict*, so as to be easily utilized in the main AVM algorithm.

3.3 AVM

The basic idea of handling string data type with AVM is considering a string as an array of characters, as shown in figure 1.

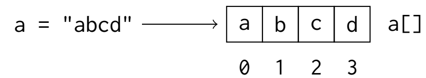

Figure 1: Array notation of a string

With the idea of thinking a string as an array of characters, the problem became simple. The problem is now reduced to finding an array of integers which satisfies all the given conditions.

3.3.1 *Environment import phase*

Firstly, in order to use the functions that are defined before the target function, an environment which the target function will be run in should be created beforehand. In this phase, external modules are also introduced to the namespace, so any functions from external module can be used inside of the target function.

The last function that are defined in the given code will be the target function, which will be processed through below steps.

3.3.2 Minimum length calculation phase

In order to guess all the characters in a string, the length of the string should be known previously. The string whose length is undefined cannot be expressed in array form, and the branch distance that we have introduced in section 3.1 could not be calculated with the undefined length string.

In this phase, the minimum length for each parameter of the target function will be calculated via the method that we have proposed in section 3.2.

3.3.3 Predicate collection phase

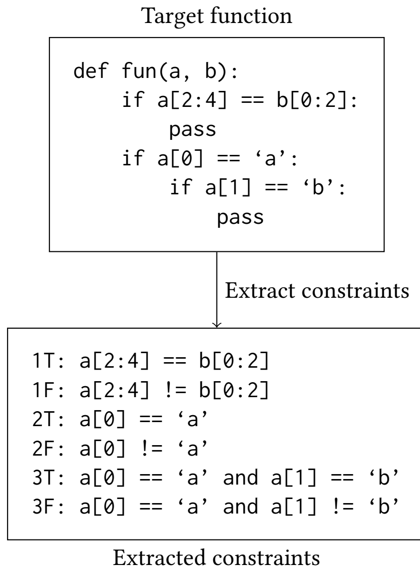

Figure 2: Extracting constrints from branches

3.3.4 AVM phase

With the basic concept of considering all string as arrays which is illustrated in figure 1, it is possible to execute AVM to the fixed size array of characters.

Exploratory move. Unlike to the integer values, an array of characters has various directions to move. In our method, we increase and decrease array for each index. For example, if `a[1]` is the next position to move, then we increase and decrease `a[1]` by 1.

After moving, we calculate branch distance for each mutations. Decreased one is our next candidate for pattern move.

Pattern move. Pattern move is done until the branch distance of the mutation stops decreasing. If the branch distance of the mutation stopped decreasing, exploratory move is redone until retry count reaches 2,000.

4 RESULTS

Baseline of evaluation is set as random string input generation. This is implemented along with the main function of this research. It tries random string inputs until it succeed to pass through the given constraints. On the other hand, our approach first starts with random input with calculated minimum length and utilizes AVM that reduces branch distance so as to do it much quicker and more efficiently.

4.1 Research Question

Main research questions of this research are as follows:
RQ1: How is ours better in speed compared to random input generation?
RQ2: How is ours better when dealing with nested constraints compared to random input generation?
RQ3: Which relational operator showed the fastest and the slowest time to be solved?

4.2 Result

Table 1 shows the result of ours compared to random string input generation. Thirty String constraints to be evaluated on are made manually for each of the six relational operator. Left-hand-sides and right-hand-sides remain the same for each operator. Figure 3 shows some examples of them.

```
if a == a:
    pass
if 'abc' == 'abc':
    pass
if a[0] == 'a':
    if a[1] == 'b':
        pass
if a[2:4] == 'ab':
    pass
if a[2:4] == b[0:2]:
    pass
```

Figure 3: Manually built string constraints

4.2.1 RQ1: How is ours better in speed compared to random input generation?

	String length		
	1	2	3
Ours	5.4	15.8	24.2
RT	13.4	641.8	5222.2

Table 1: Average number of Execution depending on string length, operator: "=="

Table 1 shows average numbers of execution for each string length regarding ours versus random input generation. Number of execution is computed by counting number of function call for ours, and for random generation technique. Only until string length of one, two and three are considered, because it sufficiently shows the trend. Ours increased almost linearly as the length increases, whereas random technique tend to increase in much higher degree.

Ops		String length		
		1	2	3
!=	Ours	3.3	12.3	22.0
	RT	11.3	362.4	4901.3
<	Ours	5.1	13.8	24.0
	RT	11.3	402.1	3821.7
<=	Ours	4.9	13.1	23.2
	RT	12.0	423.8	3812.4
>	Ours	5.2	13.9	23.1
	RT	13.3	213.6	4902.2
>=	Ours	5.4	14.0	23.7
	RT	11.8	324.5	5231.9

Table 2: Average number of Execution depending on string length for each operator

Above table 2 shows the result for the other five relational operator. In all six cases, ours overwhelmed random input generation in speed.

4.2.2 *RQ2: How is ours better when dealing with nested constraints compared to random input generation?*

Thirty nested constraints are also manually crafted. They are simply made by nesting any two atomic constraints, but intended to contain uniform operator types within them. Figure 4 shows the example.

```
if a < 'aaaaa':
    if a < 'zzzzz':
        pass
if a == 'aazaa':
    if b < 'bbbb':
        pass
    if b != 'asdf':
        pass
if a == 'cczcc':
    pass
```

Figure 4: Manually built nested string constraints

	Depth of nest		
	1	2	3
Ours	10.4	21.3	35.2
RT	19.1	TO	TO

Table 3: Average number of Execution depending on depth of nested constraints. TO(timeout)=30sec

Table 3 shows the result of ours versus random test input generation for nested constraints respectively. Depth of nest is defined as a number of constraints to pass in order to reach the deepest. Depth one means atomic constraint. Timeout is set to 30 seconds. Ours successfully found out until depth of three, while random testing only succeed in depth of one and failed from the very next. Timeout in nested constraints for random inputs is obvious result in that random input passing multiple constraints at a time rarely happens.
Ours resulted in much better performance when dealing with nested constraints compared to random input generation.

4.2.3 *RQ3: Which relational operator showed the fastest and the slowest time to be solved?*

In order to address this research question, left-hand-sides and right-hand-sides are fixed and only the operators are switched to cover all six. This can be easily done by simply aggregating numbers of execution from result of research question one and average them out, regardless of string length.

	Operator					
	==	!=	<	<=	>	>=
Ours	15.1	12.5	14.3	13.7	14.1	14.4

Table 4: Number of Execution depending on operator type

Table 4 shows the compared result regarding number of execution. Compared result tells that in case of "<, <=, >, >=," they are all similar in number of execution. On the other hand, "!=" result in the lowest number of execution and "==" yield the maximum. This implies the order of difficultness in solving the same constraint switched to each operator as follows:

- The least difficultness: "!="
- Medium difficultness: "<, <=, >, >=,"
- The highest difficultness: "=="

5 DISCUSSION / CONCLUSION

5.1 Conclusion

This paper proposes a novel approach of utilizing alternative variable method(AVM) to address test data generation for string constraints. During the work new concept of branch distance between string input and string constraint is made, and upon them successfully developed a test data generation tool.

By comparative evaluation with random string input generation, our tool shows competable result. It overwhelmed random technique in speed and performance, especially when dealing with nested constraints.

Moreover, our tool figure out the order of difficultness to search appropriate input for each of six operator.

5.2 Future works

Several improvements can be done on top of our work.

First, we have limited the operators that can be processed as mentioned in introduction. Therefore, improving and designing an branch distance for more string operation can be done. Some of the string operation can be splited into implemented string operations.

For example, substring can be implemented with our slicing, indexing, and `and` operators. There can be some demand for complicated string operations like regular expression operations.

Second, better branch distance can be designed to replace our branch distance. The branch distance which we designed has a significant problem. When we evaluated the branch distance of inequalities such as ≤, ≥, <, >, we are using exponential value of each `ord()` value of the characters. This makes the branch distance really big. For strings whose length is greater than 12, it exceeds 32-bit integer value which can be a problem. However, this is not a problem with our implementation since python supports big-integers.

Third, better algorithm can be developed to replace our AVM implementation.

Lastly, for empirical testing on real world program, the test data generator should be able to handle multiple data types. For example, we need to extend our implementation to be able to handle integer variables, and implement dynamic indices and slices. On top of that, we can handle various integer-returning string operators such as `len()`.

REFERENCES

[1] P. A. Abdulla, M. F. Atig, Y.-F. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman. Norn: An smt solver for string constraints. In D. Kroening and C. S. Păsăreanu, editors, *Computer Aided Verification*, pages 462–469, Cham, 2015. Springer International Publishing.

[2] M. Alshraideh and L. Bottaci. Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification and Reliability*, 16(3):175–203, 2006.

[3] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: A solver for string constraints. pages 105–116, 2009.

[4] P. McMinn and G. M. Kapfhammer. AVMf: An open-source framework and implementation of the alternating variable method. In *International Symposium on Search-Based Software Engineering (SSBSE 2016)*, volume 9962 of *Lecture Notes in Computer Science*, pages 259–266. Springer, 2016.

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