UDC 004.054
Doi: 10.31772/2587-6066-2019-20-1-35-39

For citation: Yakimov I. A., Kuznetsov A. S., Skripachev A. M. [Optimizing the readability of tests generated by symbolic execution]. *Siberian Journal of Science and Technology.* 2019, Vol. 20, No. 1, P. 35–39. Doi: 10.31772/2587-6066-2019-20-1-35-39

Для цитирования: Якимов И. А., Кузнецов А. С., Скрипачев А. М. Оптимизация читаемости порождаемых при символьных вычислениях тестов // Сибирский журнал науки и технологий. 2019. Т. 20, № 1. С. 35–39. Doi: 10.31772/2587-6066-2019-20-1-35-39

OPTIMIZING THE READABILITY OF TESTS GENERATED BY SYMBOLIC EXECUTION

I. A. Yakimov*, A. S. Kuznetsov, A. M. Skripachev

Siberian Federal University
79/10, Svobodnyy Av., Krasnoyarsk, 660041, Russian Federation
*E-mail: ivan.yakimov.research@yandex.ru

Taking up about half of the development time, testing remains the most common method of software quality control and its disadvantage can lead to financial losses. With a systematic approach, the test suite is considered to be complete if it provides a certain amount of code coverage. At the moment there are a large number of systematic test generators aimed at finding standard errors. Such tools generate a huge number of difficult-to-read tests that require human verification which is very expensive. The method presented in this paper allows improving the readability of tests that are automatically generated using symbolic execution, providing a qualitative reduction in the cost of verification. Experimental studies of the test generator, including this method as the final phase of the work, were conducted on 12 string functions from the Linux repository. The assessment of the readability of the lines contained in the optimized tests is comparable to the case of using words of a natural language, which has a positive effect on the process of verification of test results by humans.

Keywords: dynamic symbolic execution, natural language model, the problem of tests verification by humans.

ОПТИМИЗАЦИЯ ЧИТАЕМОСТИ ПОРОЖДАЕМЫХ ПРИ СИМВОЛЬНЫХ ВЫЧИСЛЕНИЯХ ТЕСТОВ

И. А. Якимов*, А. С. Кузнецов, А. М. Скрипачев

Сибирский федеральный университет
Российская Федерация, 660041, г. Красноярск, просп. Свободный, 79/10
*E-mail: ivan.yakimov.research@yandex.ru

Занимая около половины времени разработки, тестирование остается наиболее распространенным методом контроля качества программного обеспечения (ПО). Его недостаток может приводить к финансовым потерям. При систематическом подходе тестовый набор считается полным, если он обеспечивает определенное покрытие кода. На данный момент существует большое количество систематических генераторов тестов, направленных на поиск стандартных ошибок. Подобные инструменты порождают огромное количество трудночитаемых тестов, обладающих высокой ценой проверки человеком. Представленный в данной работе метод позволяет улучшить читаемость тестов, автоматически сгенерированных при помощи символьных вычислений, и обеспечивает качественное снижение данной цены. Экспериментальные исследования генератора тестов, включающего данный метод в качестве заключительной фазы работы, были проведены на 12-строковых функциях из репозитория Linux. Оценка степени читаемости строк, содержащихся в оптимизированных тестах, сопоставима со случаем использования слов натурального языка, что положительно сказывается на процессе верификации результатов тестирования человеком.

Ключевые слова: динамические символьные вычисления, модель естественного языка, проблема проверки тестов человеком.

Introduction. On the one hand**,** modern software systems tend to be highly complicated and expensive in development. On the other hand, software development itself is a time-consuming and error-prone process. It is very important to find serious errors before they cause any damage. Thus, in order to help developers in finding errors some bug searching methods have been developed. Software testing is one of the most popular bug searching methods. However, it is a time-consuming task that takes about a half of the development time. In order to reduce

the time expenditures associated with testing several test automation techniques have been proposed.

One of the most popular approaches to the test automation is a code-based test generation [1; 2]. Some systematic code-based test generation techniques have been developed for the last few decades. Two of them are: Search-Based Software Testing (SBST) and Dynamic Symbolic Execution (DSE). Any systematic test generation method relies on some sort of a code coverage metric. Only test data with appropriate code coverage is considered to be adequate. In order to provide required code coverage a coverage criterion needs to be defined. The popular coverage criteria are: instruction-coverage and branch-coverage. Both DSE and SBST are aimed to provide systematic code coverage for a target program.

In order to generate test cases with SBST-based tool the goal of testing needs to be defined in terms of fitness (objective) function [3]. It is convenient to use a branch coverage criterion as a goal of testing when using SBST. SBST-based tools launch target programs on some random input data. The program alternates the input data in an iterative way optimizing the value of the fitness function. Only when the fitness function is optimized the goal of testing is achieved. Final input data represents the desired test case.

DSE-based [4–6] tools maintain symbolic state in addition to the concrete (usual) state of the target program. During the execution of a target program it collects constraints on the program variables. This constraint system is called a path constraint or a PC. A PC represents an equivalence class of input data that leads the target program through the corresponding path. The execution of the target program forks on each decision point (for example conditional operator if-else) providing branch coverage. When the execution of the target program is completed, appropriate test input data can be obtained by solving the PC.

In general, code-based test generators tend to produce lots of almost unreadable test data. It is hard to verify such unreadable test data manually. This problem is called the Human Oracle Cost Problem [7]. Afshan et al. [8] proposed a method of improving readability of test cases produced by SBST-based tools. They used a character-level bigram model of a natural language to drive the search process toward more readable results. In order to do this, they added readability estimation of the target test data into the fitness function.

A character-level bigram model of a natural language is defined in terms of ordered pairs of characters, i. e. bigrams. Let (c_i, c_{i-1}) be a bigram, then P (c_i | c_{i-1}) is a probability of co-occurrence of c_i and c_{i-1}, in the language corpus, where the language corpus is a large collection of written texts. Let also $P\left(c_1^n\right)$ be a probability of belonging the whole string c_1^n of length n to the language corpus. The bigram model estimates probability $P\left(c_1^n\right)$ as shown in the equation below:

$$\hat{P}\left(c_1^n\right) \approx \prod_{i=1}^{n} P\left(c_i \mid c_{i-1}\right)$$

In order to compare strings of different length $\hat{P}$ has to be normalized always. Readability estimation N is defined as a normalized value of $\hat{P}$ as shown in equation below:

$$N\left(c_1^n\right) = \hat{P}\left(c_1^n\right)^{1/n}.$$

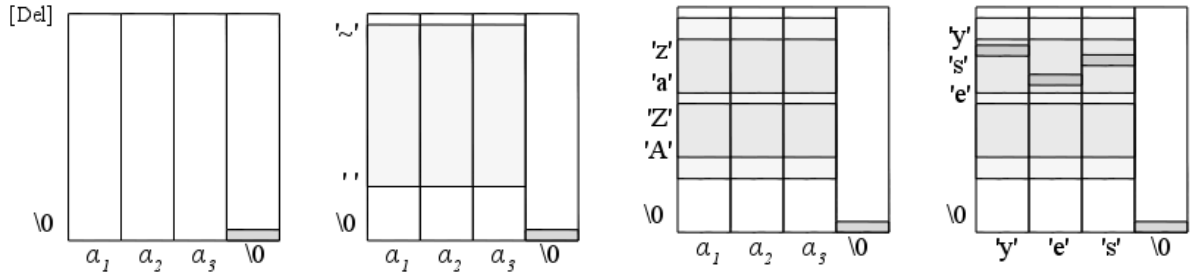
This work proposes a new method for improving readability of test data generated by DSE-based tools. In contrast to the work of Afshan et al. [8] we do not rely on any kind of fitness function. The improvement process is performed by changing a PC after the execution of the target progrem. To the best of our knowledge this is the first readability optimization method in context of the DSE.

Methods. The workflow of the proposed system includes two main stages. At the first stage, the system produces a path constraint PC, which is an abstract representation of a program state. At the second stage, the system optimizes readability of the PC constraining it with the help of a character-level bigram model. As mentioned above, every PC represents an equivalence class of some test inputs and is associated with a single program path. Some of those inputs might be more "readable" than the others. Informally speaking, the goal of the algorithm is to find "the most readable" input within the set of all possible inputs corresponding to the given PC.

Example. Let us first provide an example of the readability optimization process. Let us say we have a strlen as a target function. It takes a null-terminated string S = {'a1', a2, a3, '\0'} as an input, where eash a_i is a symbolic value and '\0' is a "concrete" terminator. The initial state of the program is represented in figure, *a*. The optimizer concretizes S transforming it into the string "yes\0". Firstly, the algorithm tries to make each a_i printable as shown in figure, *b*. Secondly, it attempts to make all of the *a-s* alphabetic as shown in figure, *c*. Finally, it attempts to rearrange a_s in an appropriate order using the bigram model. Results are shown in figure, *d*.

Memory graph. The optimizer operates on a memory graph M [9] which is an internal representation of the test data. Each node of M belongs to one of the following types: scalar (concrete or symbolic) value (*a*); concrete pointer to another node (*b*); array of nodes of a concrete length (*c*). Actually, every string S within M is represented as an array of scalar nodes.

Formal definition. A pseudocode of the algorithm is represented in Algorithm 1. The goal of the optimizer is to maximize the value of readability estimation S (N) for every string S within M. Thus, the optimizer considers only strings within M. It never violates the current PC and never changes concrete values that PC contains. The optimizer is only allowed to put constraints on symbolic values when it is safe. At the first step, it tries to make each symbolic value within S to be "alphabetic" or at least "printable". After that, it uses a bigram model in order to make the whole string being more like a "real word". During this process, every single string S within M is transformed in the following way.



a *b* *c* *d*

An example of a readability optimization algorithm work

Пример работы алгоритма по оптимизации читаемости

1. **Narrowing.** At this stage the optimizer tries to increase the readability of every symbolic character a_i within S:

– Firstly, it tries to make each a_i printable constraining it with (' ' $\le a_i \le$ '~');

– If it is successful, it then tries to make it alphabetical applying additional constraint ('A' $\le a_i \le$ 'Z' $\vee$ 'a' $\le \le a_i \le$ 'z').

2. **Concretization.** At the beginning, the optimizer focuses on the first value a_1 of the current string S. If it is possible, it tries to "assign" a random alphabetic value to the a_1 constraining it with (a_1 = *some random alphabetic character*). At the next step, the optimizer traverses through all the bigrams (a_i, a_{i+1}), where $i = 1 \ldots n - 1$, within string S. Let (a_i , a_{i+1}) be the current bigram, then:

– Firstly, the optimizer takes a concrete value of a_i. Note that if a_i is symbolic the optimizer calls the SMT-solver for its value.

– Secondly, if a_{i+1} is a symbolic value the optimizer tries to obtain the "most probable" value *val* of a_{i+1} in terms of the bigram model. If it is successful, it than attempts to apply the (a_{i+1} = *val*) constraint to a_{i+1}.

Algorithm 1 Improving readability

```
1: procedure Narrowing (Memory graph M)
2:   for all s ∈ M do
3:     if s is a string of length n then
4:       for all i ∈ {1, 2,..., n} do
5:         printable → Probe ((' ' ≤ a_i ≤ '~'))
6:         if printable = true then
7:           Probe (((('A' ≤ a_i ≤ 'Z') ∨ ('a' ≤ a_i ≤ 'z')))
8:         end if
9:       end for
10:    end if
11:  end for
12: end procedure
13: procedure Concretization (Memory graph M)
14:  for all s ∈ M do
15:    if s is a string of length n then
16:      if a_1 is symbolic then
17:        alpha ← random alphabetic character
18:        Probe ((a_1 = alpha))
19:      end if
20:      for all i ∈ {1, 2,..., n − 1} do
21:        fst ← Value (a_i)
22:        if fst is alphabetic ∧ a_{i+1} is symbolic then
23:          snd ← Next (fst)
24:          Probe (( a_{i+1} = snd))
25:        end if
26:      end for
27:    end if
28:  end for
29: end procedure
30: function Value (Scalar node a of memory graph M)
31:   if a contains concrete value then
32:     return value of a
33:   else if a contains symbolic value then
34:     Ask external SMT-solver for value of a
35:     return value returned by SMT-solver
36:   end if
37: end function
38: function Probe (Constraint e)
39:   Push a new scope into internal stack of SMT-solver
40:   PC' ← PC ∧ e
41:   if PC' is satisfiable then
42:     PC ← PC'
43:     return true
44:   else
45:     Pop the scope from internal stack of SMT-solver
46:     return false
47:   end if
48: end function
49: function Next (ASCII symbol a)
50:   b ← most probable symbol in bigram (a, b)
51:   return b
52: end function
```

Conservativeness of the algorithm. Each optimized test case leads a target program through the same path as a corresponding non-optimized version. Before applying new constraints to the current PC, the optimizer tries applying it in a fresh new scope of an external SMT-solver. If it fails, the optimizer safely pops the scope out of the stack rolling the PC back to its previous version. Only in case when the new constraint does not violate the current PC, the optimizer is allowed to apply it.

Results. In order to test the proposed system we have implemented a tool on top of the LLVM compiler infrastructure [10] and CVC4 [11] SMT-solver. We have also used a bigram model based on very large language corpora [12]. Note that before starting to work with the system a user should write a simple driver in the C-language.

Experimental results

Function	Input	Coverage	None	Basic	Bigram
strlen	[6]	5:100%	–	0.08	0.10
strnlen	[6]5	6:100%	–	0.08	0.10
strcmp	[6][6]	15:100%	–	0.04	0.05
strncmp	[6][6]5	16:100%	–	0.04	0.05
sysfs_streq	[6][6]	39:100%	–	0.05	0.07
strcpy	[6][6]	35:100%	–	0.08	0.10
strncpy	[6][6]5	36:100%	–	0.08	0.10
strcat	[10][5]	40:100%	–	0.07	0.08
strncat	[10][5]4	41:100%	–	0.07	0.08
strstr	[6][3]	19:90%	–	0.12	0.14
strnstr	[6][3]2	4:80%	–	0.09	0.10
strpbrk	[6][6]	10:80%	–	0.03	0.03

The system has been tested on 12 string-processing functions from the Linux [13] repository. Each function takes integer values and strings as an input. Experimental results are represented in table. The Input column displays an encoded format of the input data. Here notation "[n]" = $\{a_1, a_2,...,a_{n-1},'\backslash 0'\}$ represents a null-terminated string of symbolic values a_i; k represents some concrete integer. For example, strnlen "[6] 5" means that strnlen takes a single symbolic string of size 6 and an integer literal "5". The only exception is strpbrk function that takes concrete string "aeouy" as its second argument. Code coverage estimated with gcov tool is displayed in column *Coverage* in format x:y %, where x is a number of generated test cases and y is a percentage of covered instructions. Columns *None*, *Basic* and *Bigram* display average values of readability estimation N for test data generated during different experiments.

Experimental results. The results of test generation without optimization are displayed in column *None*. As the non-optimized data includes no alphabetic characters, the readability estimation is not defined in this case. The *Basic* method involves only the first, *Narrowing* phase of the optimization process. In this case, readability estimation $N \approx 0.07$ in average with standard deviation $\sigma = 0.03$. On the other hand, the *Bigram* method involves both, *Narrowing* and *Concretization* phases of the optimization. In case of *Bigram* $N \approx 0.08$ in average and $\sigma = 0.03$. Thus, the *Bigram* method shows the best results in this experimental study.

Discussion. Let us discuss the experimental results in more details on the example of the strcpy function. Strcpy function takes two string arguments - buffer A and source B. It copies data from B to A modifying A. It then returns the pointer to the modified version of A represented as A'. Thus, the format of each generated test case is (A, B) => A'. In order to give meaning to the discussion, let us suppose that the generated test cases have to be verified by a human.

Each non-optimized output contains no printable characters. Thus, we represent generated data as arrays of 8-bit integers. In this case, the real data generated with the help of CVC4 looks like: {1, 1, 1, 1, 1, 0}, {1, 1, 1, 0, 0, 0} => {1, 1, 1, 0, 0, 0}. Making sense of this data might be confusing to anyone trying to evaluate the quality of the implementation of the strcpy function. On the other hand, the so-colled Bigram method produces well-readable and less-confusing data: ("kesth", "pre") => "pre".

Configuring the optimizer. The optimizer can be configured in many ways. If the first symbol of the string is not constrained, then external SMT-solver tends to return similar results for all strings. As a consequence, without randomization of the first symbol the results look like "athes", "ath" => "ath" etc. Moreover, as the Bigram method uses the most probable values of the second characters of each bigram sometimes it tends to produce cycles like "athesthes...." etc. In order to avoid such cycles, we use the same selection algorithm as the "roulette wheel" method [14].

In addition to the bigram-based improving readability optimization we have implemented a simple "optimizer" for numeric values. In fact, numeric values generated by SMT-solvers tend to vary in a very wide range. For example, let us suppose we are testing some implementation of the quicksort algorithm. One of the generated test cases might look like: {22773760, 22773760, −2147483648, 2147483584} 4 => {−2147483648, 22773760, 22773760, 2147483584}. It is highly confusing to anyone who is trying to make a meaningful interpretation of such an unreadable test data. The optimizer incrementally tries to constraint each integer symbolic value within a memory graph M using Probe method from Algorithm 1. Let a_i be a symbolic integer contained by M. At first, the optimizer tries to apply the constraint ($-10 \le a_i \le 10$) to each a_i. If it is not successful it then tries to apply ($-100 \le a_i \le 100$) etc. The optimized version of the test case mentioned before looks like: {2, 2, −3, 5} 4 => {−3, 2, 2, 5}.

Reliability. In fact, the readability estimation of strings of different length varies in a wide range. The value of "pure" readability estimation $\hat{P}(S)$ tends to zero for very long strings. As a result, it is not reliable to compare the readability estimation of strings of different length. This negative effect is eliminated by normalization. We should further note that the goal of our research does not include the examination of the bigram model itself. However, in order to verify the readability estimation method used in the experimental study we tested it in isolation. We have tested this method using a list of Top-100 English words. The resulting readability estimation is 0, 10 which is compatible with the experimental results. Finally, the reliability of the experimental results achieved by any

code-based test generator depends on the provided code coverage. In the given experiments the instruction coverage is 95% in average and it is 100 % for 9 functions. In case of functions with non-100 % coverage the NULL-returning branch is not covered. We can confidently say that the obtained results are reliable enough.

Conclusions. This work introduces a new method of readability optimization in context of Dynamic Symbolic Execution based on a natural language model. This method has been successfully examined against 12 string-processing functions from the Linux repository. The experimental results show that this algorithm significantly improves the readability of automatically-generated test data. The readability of the optimized test cases is compatible to the readability of human-written texts. Developers who manually verify generated test data would take advantage of using this method.

References

1. Anand S., Burke E. K., Tsong Yueh Chen et al. An Orchestrated Survey of Methodologies for Automated Software Test Case Generation. *Journal of Systems and Software*. 2013, Vol. 86, No. 8, P. 1978–2001. Doi: 10.1016/j.jss.2013.02.061.

2. Cadar C., Godefroid P., Khurshid S. et al. Symbolic Execution for Software Testing in Practice: Preliminary Assessment. *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, 2011, P. 1066–1071. Doi: 10.1145/1985793.1985995.

3. Tracey N., Clark J., Mander K. et al. An automated framework for structural test-data generation. *Proceedings of the 13th IEEE International Conference on Automated Software Engineering*. 1998, P. 285–288. Doi: 10.1109/ASE.1998.732680.

4. Cadar C., Ganesh V., Pawlowski P. M. et al. EXE: Automatically Generating Inputs of Death. *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS '06)*. ACM, New York, 2006, P. 322–335. Doi: 10.1145/1180405.1180445.

5. Godefroid P., Klarlund N., Sen K. DART: Directed Automated Random Testing. *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 2015, P. 213–223. Doi: 10.1145/1064978.1065036.

6. King J. C. Symbolic Execution and Program Testing. *Communications of the ACM*. 1976, Vol. 19, No. 7, P. 385–394. Doi: 10.1145/360248.360252.

7. Barr E. T., Harman M., McMinn P. et al. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering*. 2015, Vol. 41, No. 5, P. 507–525. Doi: 10.1109/TSE.2014.2372785.

8. Afshan S., McMinn P., Stevenson M. Evolving Readable String Test Inputs Using a Natural Language Model to Reduce Human Oracle Cost. *IEEE the 6th International Conference on Software Testing, Verification and Validation*. 2013, P. 352–361. Doi: 10.1109/ICST.2013.11.

9. Sen K., Marinov D., Agha G. CUTE: A Concolic Unit Testing Engine for C. *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-13)*. 2005, P. 263–272. Doi: 10.1145/1095430.1081750.

10. Lattner C., Adve V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, 2004, P. 75.

11. Barrett C., Conway C. L., Deters M. et al. CVC4. *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 2011, P. 171–177.

12. Jones M. N., Mewhort D. J. K. Case-sensitive letter and bigram frequency counts from large-scale English corpora. *Behavior Research Methods, Instruments and Computers*. 2004, Vol. 36, No. 3, P. 388–996.

13. Linus Torvalds et al. Linux kernel source tree. Available at: https://github.com/torvalds/linux (accessed: 20.11.2018).

14. Lipowski A., Lipowska D. Roulette-wheel selection via stochastic acceptance. *Physica A: Statistical Mechanics and its Applications*. 2012, Vol. 391, No. 6, P. 2193–2196. Doi: 10.1016/j.physa.2011.12.004.

Yakimov Ivan Aleksandrovich – Senior lecturer; Institute of space and informational technologies, Siberian Federal University. E-mail: ivan.yakimov.research@yandex.ru.

Kuznetsov Aleksandr Sergeevich – Cand. Sc., Assistant professor; Institute of space and informational technologies, Siberian Federal University. E-mail: ASKuznetsov@sfu-kras.ru

Skripachev Anton Mikhailovich – Master's degree student; Institute of space and informational technologies, Siberian Federal University. E-mail: skram@list.ru.

Якимов Иван Александрович – старший преподаватель; Институт космических и информационных технологий, Сибирский федеральный университет. E-mail: ivan.yakimov.research@yandex.ru.

Кузнецов Александр Сергеевич – кандидат технических наук, доцент; Институт космических и информационных технологий, Сибирский федеральный университет. E-mail: ASKuznetsov@sfu-kras.ru.

Скрипачев Антон Михайлович – магистрант; Институт космических и информационных технологий, Сибирский федеральный университет. E-mail: skram@list.ru.