Mining constant information for readable test data generation

Mingzhe Zhang, Yunzhan Gong, Yawen Wang* and Dahai Jin

State Key Laboratory of Networking and Switching Technology,
Beijing University of Posts and Telecommunications,
Beijing 100876, China
Email: zmz420@126.com
Email: gongyz@bupt.edu.cn
Email: wangyawen@bupt.edu.cn
Email: jindh@bupt.edu.cn
*Corresponding author

Abstract: Automated test data generation tools produce test data that can achieve high coverage faster than test data generated manually by a tester. However, the test data generated by automated tools has been shown to not help developers find more bugs. The main reason is that it is difficult for human testers to understand and evaluate the test data. In this paper, an approach is introduced to automatically generate readable test data, which has been implemented in a tool called CTS. CTS can mine constant information from projects under testing and obtain heuristic information by aggregating and rating related constants. CTS adds heuristic information to the automatic test data generation process to generate test data that is quick and easy for a human to comprehend and check. Empirical experiments show that the proposed approach can improve the efficiency of test data generation and generate test data that is more convenient for a human oracle.

Keywords: test data generation; readable test data; constraint-based testing; CBT; symbolic execution.

Reference to this paper should be made as follows: Zhang, M., Gong, Y., Wang, Y. and Jin, D. (2021) 'Mining constant information for readable test data generation', *Int. J. Embedded Systems*, Vol. 14, No. 1, pp.9–18.

Biographical notes: Mingzhe Zhang received his BE in Computer Science and Technology from the Shandong Normal University, Jinan, in 2012. He is currently a PhD candidate in the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing. His current research interests mainly focus on software testing and program analysis.

Yunzhan Gong received his PhD in Computer Science from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, in 1991. He is currently a Professor and a Supervisor of Doctoral students in the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing. His research interests include fault tolerant computing and software testing.

Yawen Wang received her PhD in Communication and Information System from the Beijing University of Posts and Telecommunications, Beijing, in 2010. She is currently an Associate Professor in the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing. Her research interests include static analysis and automated software testing.

Dahai Jin received his PhD in Information Security from the Armored Engineering Institute of the PLA, Beijing, in 2006. He is currently an Associate Professor in the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing. His research interests include software testing and static analysis.

1 Introduction

Software testing is an important activity in software development and is widely used to assure software quality (Panda and Mohapatra, 2017; Pan et al., 2018; Neves et al., 2016). However, it is also laborious and time-consuming, and programmers may spend a quarter of their work time solely on developing testing (Beller et al., 2015; Daka and Fraser, 2014). Since test data generation is the most challenging and time-consuming process in software testing, several automated test data generation tools have been proposed to reduce the time developers need to spend on testing.

Automated test data generation approaches mainly focus on code coverage as a unique goal without accounting for other relevant factors (McMinn et al., 2010; Palomba et al., 2016). As a consequence, the test data generated by these approaches usually provides no measurable improvement in the number of bugs actually found by manual testing (Fraser et al., 2015; Shamshiri et al., 2015; Ceccato et al., 2015). The automated generated test data usually needs a human oracle due to the frequent non-existence of an automated oracle. In addition, the automated generated test data are often arbitrary-looking, and arbitrary-looking test data are both difficult to comprehend and time-consuming to check. Two studies (Fraser et al., 2015; Fraser and Arcuri, 2013) reported that developers spend up to 50% of their time comprehending and checking arbitrary-looking unreadable machine-generated test data.

The main reason why automated generated test data is unreadable is because automated test data generation approaches do not incorporate knowledge about the input into the generation process. For example, some programs use ‘month’, ‘day’, and ‘year’ as input variables, but if the generation process does not take into account what realistic date test data looks like, then the test data will be difficult to understand and check. It is necessary to incorporate domain knowledge into the process of test data generation, such as that the domain knowledge of ‘month’ is ‘month’ is between 1 and 12. However, since formal specifications and domain knowledge are frequently unavailable to a program under testing, extracting more information from the code becomes even more critical.

Constraint-based testing (CBT) (Offutt and DeMilli, 1991; Boonstoppel et al., 2008) is one of the approaches used to generate test data automatically. The general idea behind CBT is to extract constraints from a program and exploit constraint solving to generate test cases for the program (Gotlieb, 2015). Symbolic execution (Mishra et al., 2005) is usually used for collecting constraints, as it interprets a program by using symbolic inputs along a path to compute a constraint called the path constraint. For the CBT approach, incorporating knowledge about the input domain in the generation process would mean adding extra constraints to the program constraints or designating a start point to seed the generation of test data. For example, in the case that a program uses dates as input variables, if the program constraints do not include any constraints that restrict the value of ‘month’ to a readable value (we call variables that lack constraints ‘under-constrained variables’), the extra constraint of ‘month’ is 1 <= month <= 12.

Under-constrained variables are common in real-world projects (details in Subsection 3.2.1). Due to the lack of constraints, under-constrained variables need to be solved in a large solution space to generate test data and are prone to generating unreadable test data. Compared with under-constrained variables, variables with integrated constraints reduce the solution space, but determining how to generate readable test data in the solution space is also an important problem that needs to be solved. As a result, both obtaining extra constraints and providing a starting point to seed test data generation are important issues that must be addressed in generating readable test data based on the CBT approach.

In this paper, we present an approach to alleviate unreadable test data problems for numerical data types. Rather than relying on frequently unavailable program specifications and expert knowledge to generate readable test data, we leverage constant information mined from the source code to generate extra constraints and seed the generation process. We select four types of specific operators and use these specific operators to mine constants that are related to the input variables. The mined constant information is then processed and used to generate extra constraints and seed the generation process. We implement our approach in a tool called code testing system (CTS). CTS is an automated unit testing tool for C code, and it can automatically generate test data for functions under testing.

We present an empirical study in which we evaluated the capabilities of our approach. We evaluated CTS on the real-world project *double*. The evaluations show that our approach can improve the efficiency of test data generation and generate test data that is more convenient for a human oracle.

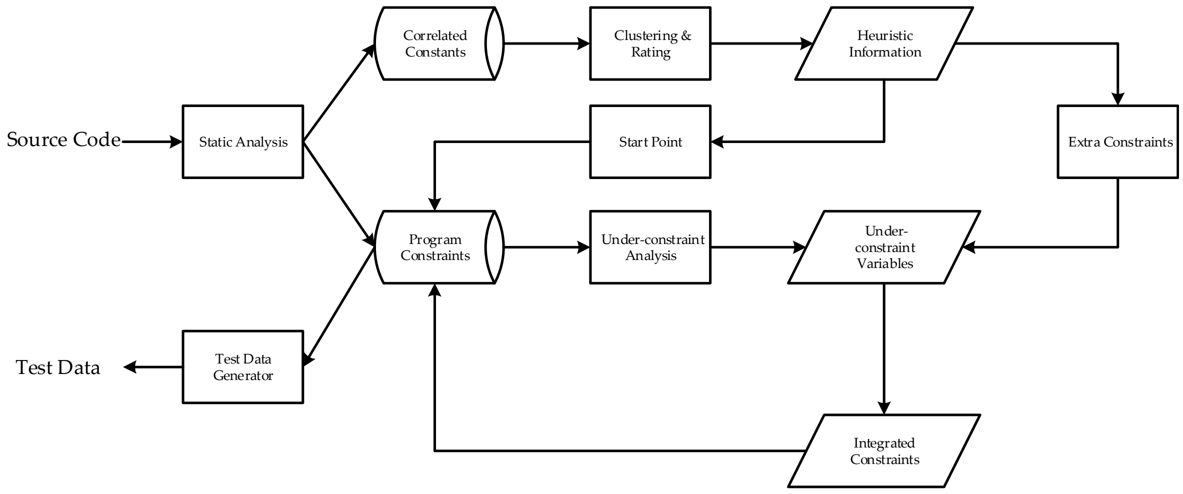
The contributions of this paper are as follows:

1 We divide the input variables into integrated constraint variables and under-constrained variables, and we point out that under-constrained variables are a direct cause of tools generating unreadable test data. To negate the effect of under-constrained variables, we reduce the generation of unreadable test data by adding extra constraints.

2 We propose a method for generating heuristic information by mining constant information. By aggregating and rating the constants, we can obtain heuristic information for generating readable test data.

The rest of this paper is organised as follows: Section 2 introduces the background and basic idea of our approach. Section 3 describes the details of our approach implementation. Section 4 presents the experimental results. Section 5 discusses related work. Section 6 gives a conclusion.

Figure 1 Workflow of our approach



2 Background and basic idea of our approach

In the CBT approach, test data is a set of data that has satisfied the constraints of the program. As a result, the test data can cover the code that corresponds to constraints that the test data has satisfied. Since an automated oracle is not always available, the test data may be frequently evaluated manually; it is not enough for the test data to only satisfy program constraints. For human oracle processes, it is important to have a set of readable test data for programs under testing. Some programs make use of human-recognisable variables as inputs; additionally, the data types of input variables are different, and consequently, the requirements for the readability of test data vary according to program characteristics and the data types of the inputs. For example, string inputs usually need data with a specific structure and realistic semantics, such as the test data for a country name, email address or URL. For numerical data types, the readability must qualify in two respects:

2.1 Value readability

The values of readable numerical data should be easily comprehensible or easy to check, meaning that the value ranges of readable numerical data can be handled by humans. In the CBT approach, test data is constrained by the program constraints. Satisfying program constraints is the most basic requirement for test data. On the basis of satisfying the program constraints, the readable test data should be reasonable for the program under testing and should be able to be handled by humans. There are some under-constrained variables in real-world projects. For these under-constrained variables, the test data generated based on the existing constraints only meets the most basic requirements. It is hard to determine a readable value in an unconstrained solution space; as a result, adding extra constraints to the under-constrained variables becomes necessary.

2.2 Semantic readability

In some real-world projects, numerical data type inputs are human-recognisable variables, such as ISBNs and zone improvement plan (ZIP) codes. Test data that meets structural requirements or have a specific meaning are semantically readable. Two approaches can be used to generate semantically readable test data. One approach is to add extra constraints to the program constraints according to the specific semantic requirements, but the extra constraints are difficult to obtain through automated analysis and can be added by manual analysis. However, manual analysis would require expert knowledge because domain knowledge is required to convert information in a program into extra constraints; therefore, the conversion process requires significant human effort. The other approach is to acquire test data directly from an external knowledge base such as (Bozkurt and Harman, 2011), in which they acquired test data from compositions of many existing web services. This approach requires the availability of an external knowledge base and the need to establish a relationship between program elements and external knowledge bases. Finding an available external knowledge base and establishing a relationship require much human effort. Furthermore, there are few available external knowledge bases, which makes for poor scalability.

In this paper, we focus only on generating value-readable test data for numerical data types in real-world projects. The key ideas of our approach are to use the constant information in the projects under testing to generate extra constraints and obtain a start point; use extra constraints to constrain the under-constrained variables to generate value-readable test data; and use the start point to seed the test data generation. This approach consists of three phases. First, program constraints and related constants (constants that act directly on input variables through specific operators; details can be found in Subsection 3.1) are extracted. Then, we analyse the extracted data to obtain under-constrained variables and heuristic information, use the heuristic information to generate extra constraints and obtain a start point, and finally generate readable test data by querying the start point and solving the integrated constraints. Figure 1 shows the workflow of our approach.

3 Approach

In this section, we describe the details of the readable test data generation for numerical data types by CTS.

3.1 *Extracting program constraints and related constants*

CTS extracts information (program constraints and related constants) from the source code via static analysis. In the following, we first introduce the concepts of program constraints and related constants and then present the details of static analysis.

- *Program constraints:* Program constraints are a set of conditions in one path that input variables must satisfy; they can be obtained by symbolic execution techniques. Taking the branch statement if ($a > 10$) as an example, supposing that a_sym is the symbol of a, after the branch statement is executed by the symbol, we can obtain the following constraint on variable a: $a\_sym > 10$.
- *Related constants:* Related constants are constants that act directly on input variables through specific operators (including ==, >=, <=, >, <, etc. the specific operators are listed in Subsection 3.2.2). For example, in the expression $a > 10$, supposing that a is an input variable, then 10 is a related constant of a.

Before compilation, the source code is just character streams, and it is difficult to extract information from character streams. To perform static analysis of the source code, we convert the source code into structural data such as an abstract syntax tree (AST) and control flow graph (CFG).

After CTS obtains the AST and CFG of the program under testing, CTS uses symbolic execution techniques to execute the program and obtain program constraints. Symbolic execution is performed by traversing the CFG along the given path and simulating the effect of executing the code contained in a node of the AST that corresponds to the node in the CFG. The constraints can be found from the symbolic execution results of the branch statements; the program constraints are the conjunction of all constraints in the path.

The AST is a tree representation of the abstract syntactic structure of the source code. Each node of the tree denotes a construct occurring in the source code. We can obtain the relationship among variables, operators, and constants from the AST; related constants are then extracted by analysing the relationship.

To obtain a related constant, we first need to determine the CFG nodes in need of further analysis. Since each type of statement has a corresponding type of CFG node, we select three types of statements that may contain related constants: branch statements, declaration statements and assignment statements; thus, the corresponding CFG nodes are the target nodes we need to analyse. When these CFG nodes contain constants that satisfy the definition of related constants, the resulting constants must be the related constants. Second, for a given path, we find all the target CFG nodes and then traverse the corresponding AST nodes of the target CFG node to find the related constants. Finally, we store the related constants and the corresponding operators and input variables.

Code 1

```
1 int f (int a[], int x){
2 x = strlen(a);
3 if (x >= 3){
4 a[x] = 1;
5 }
6 …
7 }
```

Taking Code 1 as an example, the CFG and AST of Code 1 are shown in Figure 2 for a given path: 0-1-2-3-5-6 (the numbers represent the CFG edges in Figure 2). We first determine the CFG nodes that need to be analysed: stmt_1, if_head_2, and stmt_3. Then, we traverse the corresponding AST nodes and find that the AST nodes corresponding to if_head_2 contain the special operation >=. The operands of the special operation >= are input variable x and constant 3. 3 is the related constant of x.

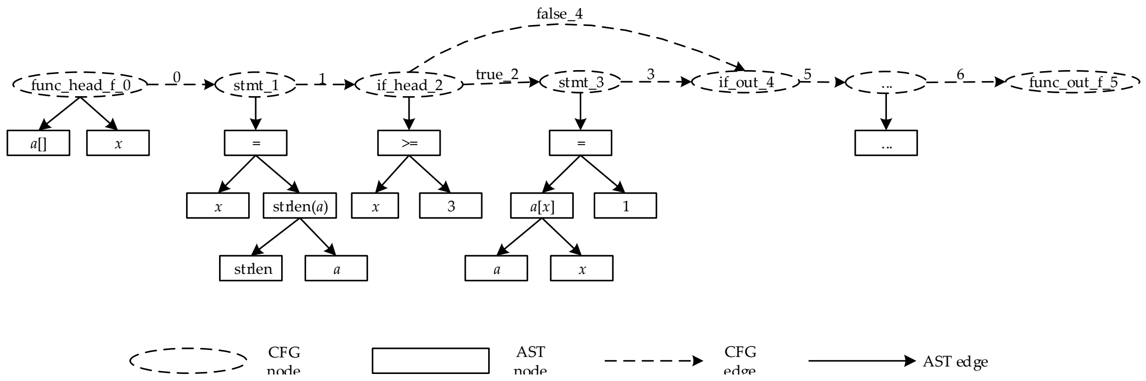
3.2 *Analysing the data that has been extracted*

3.2.1 *Under-constrained analysis*

The CBT approach generates test data by solving constraints of input variables. When the constraints of input variables are incomplete, the constraint solver needs to generate test data in a large solution space, which may produce test data that is difficult for humans to process (for example, the values of the test data are too large), resulting in poorly readable test data.

- *Under-constrained variables:* An under-constrained variable is a variable whose constraint either has no upper or lower bound or no bound at all. For example, supposing that x is an input variable of the function under testing, for a given path, the constraint of x is $x > 10$; in this example, x does not have an upper bound of the constraint. If x is an int type (referring to the C language program), x will take a value between 11 and 32,767 under the current constraint. If the value of x is too large, it will be difficult for a human oracle to process.
- *Integrated constraint variables:* An integrated constraint variable is a variable whose constraint has both an upper and lower bound.

Figure 2 AST and CFG of Code 1



To understand the distribution of under-constrained variables in real-world projects, we have calculated the distribution of under-constrained variables in the real-world project *double*. *Double* is a double-precision mathematical function library, available from Astronomy and Numerical Software Source Codes (http://www.moshier.net/). We verify that a variable is under-constrained by analysing the obtained program constraints. The results are shown in Table 1.

Table 1 The distribution of under-constrained variables in *double*

Project	*Double*
Number of functions	277
Number of paths	978
Number of paths with under-constrained variables	734
Number of paths without under-constrained variables	104
Number of infeasible paths	140

The CBT approach is path-oriented. For different paths in a function, the constraints of the input variables may be under-constrained or integrated, and as a result, we cannot determine whether an input variable is under-constrained or integrated in a function and can only more narrowly determine whether the input variable is under-constrained or integrated for a given path. In our analysis of *double*, we counted the number of paths with under-constrained variables (a path here refers to a path in the function, and the number of paths in Table 1 refers to the sum of the paths of a function in *double*). According to the results in Table 1, up to 75% of the paths in *double* contain under-constrained variables. If we do not add extra constraints to these variables, the program will generate a large number of test data that a human oracle may have difficulty processing.

3.2.2 Aggregating and rating related constants to generate heuristic information

After we obtain the related constants in the program, we must analyse them. Related constants act on input variables through various operators that have different effects on the input variables. According to the semantics and usage scenarios of these operators, the relationship between input variable values and constants can be determined by backward reasoning, and related constants are used to inspire and constrain the generation of test data. In this section, we first list all the specific operators involved in the extraction of the related constants and explain why these specific operators were selected. Then, we show how to generate the heuristic information by aggregating and rating related constants according to the semantics and usage scenarios of specific operators.

3.2.2.1 Specific operators in related constant extraction

Equal to (==)

When the input variable and constant are related by ==, this indicates that the constant is one of the target values of the input variable. Furthermore, since the constant is set by the developer according to a certain specification, the constant is reasonable for both the input variable and the program under testing. The constant can be used directly to seed test data generation for the input variable and its related variables.

Greater than or equal to (>=) and less than or equal to (<=)

These two operators not only compare input variables and constants but also provide boundary values for input variables. Because the input variable can be equal to the related constant under the current operation, the related constant can be used as the boundary value of the input variable. In addition, since the constant is set by the developer according to a certain specification, it can also be used to seed test data generation.

Function call (())

When the argument of the called function contains a constant, the constant is the related constant of the corresponding parameter. For example, supposing that function *f*(int *x*, int *y*) is called as *f*(*a*, 5), *a* is a variable in the calling function, and 5 is the related constant of *y*. Since the related constant obtained by the function call is set by the developer, the value of the related constant should be reasonable to them; thus, we can also use this value to seed test data generation.

Greater than (>) and less than (<)

When a constant acts on an input variable through these operators, they indicate that the constant is comparable to the input variable. If the expected value of the input variable is much larger than the compared constant, then the comparison will become meaningless. Based on this reasoning, the value of the input variable under the current operation can be generated around the value of the related constant. Because the related constant is set by the developer according to a specification, the related constant is used to heuristically generate data that meet the requirements of the program and developers.

Table 2 Related constant rates in real-world projects

Project	*LOC*	*Input variables*	*Input variables with related constants (rate)*	*Functions*	*Functions with related constants (rate)*
128 bit	19,655	295	60 (20%)	204	51 (25%)
Single	17,682	335	59 (18%)	170	48 (28%)
Double	37,452	608	221 (36%)	246	144 (58%)

We collect and extract related constants based on the above operators and use the extracted related constants to generate readable test data. We applied the above operators to the extraction process of related constants and conducted an empirical analysis of real-world projects to see how many related constants could be obtained. Table 2 shows the results of our analysis. All the projects we used can be found at Astronomy and Numerical Software Source Codes; most of the inputs of these projects are numerical data types. As shown in Table 2, the rates of related constants vary in different projects.

3.2.2.2 Aggregating and rating related constants

When we obtain all the related constants in a project, we must analyse the related constants to generate the heuristic information. Not every variable has related constants – for instance, under-constrained variables may not have related constants. To generate heuristic information for all the under-constrained variables, we aggregate the related constants and generate heuristic information to produce extra constraints by analysing the aggregation results. We then rate the related constants by rating the specific operators and select the related constants with strong heuristics to seed the test data generation.

Aggregating related constants

Under-constrained variables may not have related constants, so we must generate heuristic information based on existing related constants. The number of related constants obtained by a single type of specific operator may be too small, thereby making it difficult to obtain valid information from these related constants. As a result, we aggregate all related constants to extract heuristic information.

To obtain all the related constants in the project under testing, we aggregate the related constants obtained by each type of operator. Before analysing the aggregated data, we first clean the data for the purpose of removing outliers. Tukey's test (Vardy and Moller, 2005) is used for data cleaning.

First, we have to calculate the first quartile ($Q1$) and the third quartile ($Q3$). $Q1$ is defined as the middle number between the smallest number and the median of the related constant data set; $Q3$ is the middle value between the median and the highest value of the related constant data set.

When the data set of related constants is arranged in ascending order, $Q1$ is given by equation (1), where r_c denotes a related constant (same below).

$$Q1 = \left(\frac{n+1}{4}\right)^{\text{th}} rc \tag{1}$$

$Q3$ is given by equation (2).

$$Q3 = \left(\frac{3(n+1)}{4}\right)^{\text{th}} rc \tag{2}$$

After obtaining $Q1$ and $Q3$, we can use equation (3) and equation (4) to calculate the upper limit (ul) and lower limit (ll) of the normal related constants.

$$ul = Q3 + 1.5\,(Q3 - Q1) \tag{3}$$

$$ll = Q1 - 1.5\,(Q3 - Q1) \tag{4}$$

Any related constant not included between ul and ll is an outlier; we remove all outliers from the related constant data set to complete the data cleaning. We cleaned the data of the three projects presented in Table 2 and found that the outliers are related constants that appear either once or twice. Based on this result, we will try to remove related constants with lower frequency to simplify the data cleaning process in future experiments.

When the data cleaning is complete, we can obtain the upper and lower limits (ul and ll) of the related constants. Excluding outliers, the related constants are distributed between ul and ll. Since all the related constants are set by the developer according to specifications, the aggregated data can, to a certain extent, reflect the expected values of the project under testing. Because an under-constrained variable has no upper or lower bound, we need to choose values between ul and ll as the constraint boundaries. The values should be points at which the balance is in equilibrium between ul and ll. However, the frequency of each related constant varies in the related constant data set, so we use the weighted arithmetic mean as the value. The weighted arithmetic mean can be calculated using equation (5), where f denotes the frequency of the related constant.

$$\overline{c} = \frac{\sum_{i=1}^{n} f_i rc_i}{\sum_{i=1}^{n} f_i} \tag{5}$$

We use $\overline{c}$ as a constraint boundary to add an extra constraint for under-constrained variables. Since we do not know the relationship among existing constraint boundaries, $\overline{c}$, *ul* and *ll*, we need to determine the relationship by comparison and generate the corresponding extra constraints. Supposing that x is an under-constrained variable, Table 3 shows how to add an extra constraint to the under-constrained variable x, and the current constraint is the constraint of x for a given path. If the current constraint does not intersect with the interval formed by *ll* and *ul*, we increase or decrease the step value d to the existing constraint boundary to form an integrated constraint. The step value d can be given by the user or set to a fixed value. For other under-constrained variables in the project under testing, we can add extra constraints according to the rules in Table 3 to generate integrated constraints. When we finish generating the integrated constraints for the under-constrained variables, we use a constraint solver (Xing et al., 2014) to solve the constraints and generate readable test data.

Table 3 Adding extra constraints to the under-constrained variables

Type	*Current constraint*	*Relationship among c1, c2, $\overline{c}$, ll and ul*	*Extra constraint*	*Integrated constraint*
Missing upper bound of the constraint	$x > c1$	$c1 < \overline{c}$	$x < \overline{c}$	$c1 < x < \overline{c}$
	$x > c1$	$\overline{c} < c1 < ul$	$x < ul$	$c1 < x < ul$
	$x > c1$	$c1 > ul$	$x < c + d$	$c1 < x < c1 + d$
Missing lower bound of the constraint	$x < c2$	$c2 > \overline{c}$	$x > \overline{c}$	$\overline{c} < x < c2$
	$x < c2$	$ll < c2 < \overline{c}$	$x > ll$	$ll < x < c2$
	$x < c2$	$c2 < ll$	$x > c2 - d$	$c2 - d < x < c2$
Missing constraint	x	N/A	ll < x < ul	$ll < x < ul$

Rating related constants

We have provided four types of specific operators in the above section. Due to the differences in the semantics of each type of operator, the heuristic ability of the corresponding related constants also varies. For example, the related constants obtained by function call are values set by the developer, and these values are reasonable for the developer. The related constants obtained by greater than (>) or less than (<) indicate that the related constants and the corresponding input variables are comparable; we speculate that the input variable values may be close to the related constants, but this is only a conjecture and may not be accurate.

According to the semantics of each type of specific operator and the heuristic ability of the corresponding related constants, we divide the specific operators into two groups: strong heuristics and weak heuristics. Strong heuristic operators include equal to (==), greater than or equal to (>=), less than or equal to (<=), and function call (()). Weak heuristic operators include greater than (>) and less than (<). The corresponding related constants are strong heuristic-related constants and weak heuristic-related constants.

We collect all the strong heuristic-related constants as the start point to seed the generation and index the related constants for test data generation. For an integrated constraint variable, we first query whether the variable has strong heuristic-related constants and determine if it satisfies the constraints. If the constants satisfy the constraints, we will use the constants as the test data. If the constants do not satisfy the constraints, we will use the constants to increase or decrease the fixed value to try to satisfy the constraints and use the value that satisfies the constraints as the test data.

4 Evaluation

The approach is integrated into CTS, which is an automated unit testing tool for C code. To evaluate our approach, CTS is divided into two versions: CTS with related constant analysis and CTS without related constant analysis (CTS-Basic). The objective of the experiments was to investigate the following research questions.

RQ1 Does our approach help CTS improve the efficiency of test data generation?

To answer RQ1, we evaluated the test data generation times of CTS and CTS-Basic.

RQ2 Does our approach help CTS generate test data that is more convenient for a human oracle?

To answer RQ2, we compared the readability of the test data generated by CTS and CTS-Basic.

The experiments are conducted on the real-world project *double*. We use a simple random sampling method to select 25 of the functions in *double* that contain under-constrained variables. For these functions, we run CTS and CTS-Basic to generate test data and record the time and test data. The experiments in the evaluation were run on a laptop with a 2.20 GHz CPU and 8 GB RAM running an Ubuntu-14.04 operating system.

Figure 3 Test data generation times of CTS and CTS-Basic (see online version for colours)

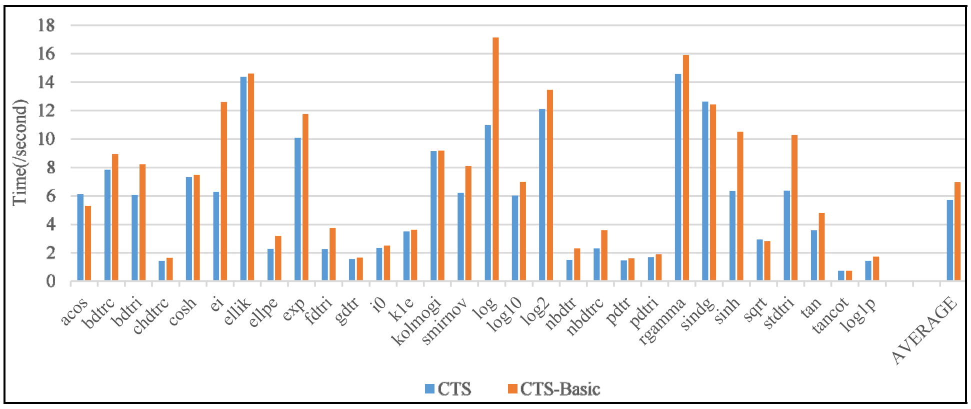
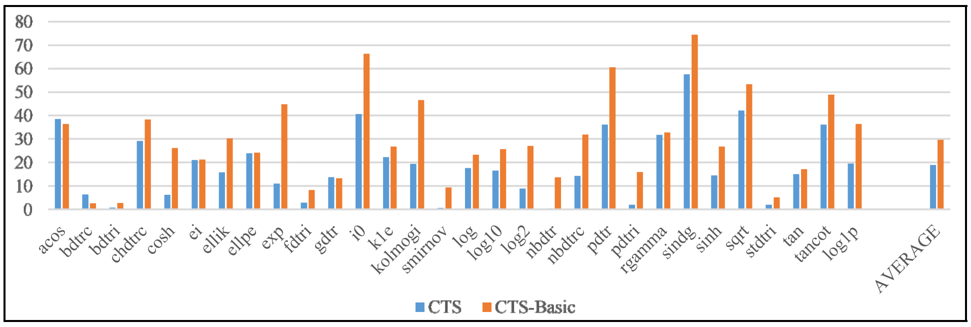


Figure 4 Means of test data (floating-point data types) of CTS and CTS-Basic (see online version for colours)



4.1 RQ1

The results of the test data generation times of CTS and CTS-Basic are shown in Figure 3. Figure 3 shows that for most of the functions under testing, the test data generation time of CTS is shorter than that of CTS-Basic. This is because CTS-Basic does not add extra constraints to under-constrained variables, and the constraint solver needs to generate test data in a large solution space, which leads to a longer generation time for CTS-Basic. To generate integrated constraints for under-constrained variables, CTS obtains heuristic information by mining and analysing related constants and uses the heuristic information to generate extra constraints or seed test data generation. The integrated constraints generated based on heuristic information not only reduce the solution space but also constrain the test data within a reasonable interval. As a result, we can obtain test data in a shorter time by using CTS.

4.2 RQ2

Since the main purpose of our approach is to generate readable test data, we need to evaluate the generated test data. For test data of numerical data types, there is no uniform standard to determine what kind of test data is readable. However, because we generate readable test data for the convenience of a human oracle, for humans, the smaller the size of the data, the more convenient it is to calculate (in the case where the absolute values of the compared data is greater than 1 and the numbers of digits are similar, not 1.1 and 1.00000001, or 0.01 and 0.0000001). In this experiment, we determine whether the test data is convenient for a human oracle by comparing the size of the generated test data.

$$\overline{t} = \frac{\sum_{i=1}^{n} |t_i|}{n} \tag{6}$$

To compare the size of the generated test data, we use equation (6) to calculate the mean of the test data for the function under testing, where t denotes test data. In this experiment, we focus only on floating-point test data.

As shown in Figure 4, the test data generated by CTS is significantly smaller than those generated by CTS-Basic. These results indicate that adding extra constraints based on heuristic information can effectively limit the size of the test data. Compared with the test data generated by CTS-Basic, the test data generated by CTS is smaller and more convenient to calculate. Therefore, for developers, the test data generated by CTS is more readable than those generated by CTS-Basic. The mean value generated by CTS is greater than that generated by CTS-Basic for the three functions *acos*, *bdtrc*, and *gdtr*. There are two reasons for this. First, the constraint of an under-constrained variable may not intersect with the interval formed by *ll* and *ul*; this causes the variable to generate test data in a larger interval. Second, our constraint solver randomly takes values within the interval corresponding to the integrated constraint. Although CTS-Basic lacks constraints, it is possible for CTS-Basic to generate smaller test data than CTS.

4.3 Threats to validity

Threats to internal validity are concerned with possible bugs in the implementation of our approach. To reduce these threats, we performed a manual check to review all the code we produced for correctness.

Threats to external validity regard the potential bias in the selection of projects used in the evaluation. These threats are related to the following question: were the programs we selected representative? To reduce these threats, we tried to remove any bias related to the selection of samples by adopting a third-party benchmark. However, there is a chance that the samples are still not representative.

Threats to construct validity are concerned with whether our measurements reflect real-world situations. In our study, we used criteria to measure the performance of our approach, including the size of the generated test data and the test data generation time. However, these two criteria do not take the human costs (such as determining how to aggregate and rate related constants) into account.

5 Related work

The test data generated by an automated test data generation approach is usually unrealistic and unreadable (Fraser et al., 2015; Fraser and Arcuri, 2013), and approaches have been presented to deal with this problem.

Some approaches that incorporate knowledge about the input of the program into the automated test data generation process are used to generate readable test data, and the knowledge can be extracted from experts, source code, the web and so on. Shahbaz et al. (2012) presented an approach for generating values for string data types by finding valid string inputs on the web. However, the identifiers used to generate web queries must be meaningful. The identifiers in an object-oriented program are usually meaningful, but in a C program, a meaningful identifier is hard to find. A similar approach was presented by McMinn et al. (2012) to extract knowledge from the internet and use the knowledge as string inputs. The method of splitting identifiers into constituent words used in this paper is based on underscoring and camel casing. Bozkurt and Harman (2011) acquired test data from the compositions of many existing web services. Their work can only deal with programs that require structural and semantic data as input, for example, using ISBNs or ZIP codes as program inputs.

Afshan et al. (2013) presented an approach in which a natural language (NL) model is incorporated into a search-based input data generation process, and the NL model is used to assign a probability score to string test data, with the score ranking the readability of the generated string test data. However, the NL model requires various types of text to train the model to deal with any string. McMinn et al. (2010) extracted constraints from the program under testing; these constraints were extracted based on sanitisation routines or defensive programming constructs and used to ‘correct’ automatically generated inputs. However, not every program has such constraints.

Because the test data values produced by automated test data generators are arbitrary-looking, these values are difficult to understand and maintain. Some works have sought to improve the comprehensibility of test data or reduce the workload of testers. Panichella et al. (2016) proposed an approach that automatically generates test case summaries of programs to improve understandability. They extracted information from code and code comments, including verbs, nouns, and prepositional phrases that can be expanded to generate readable NL sentences. However, this work requires the quality of the code to be good enough; if not, there will not be enough information to generate the summaries. Li et al. (2016) presented an approach to automatically generate NL documentation of unit test cases to ameliorate the burden of maintaining unit test cases for developers. Since reducing the workload of testers can give testers more time to comprehend unreadable test cases, improving the understandability is also widely related to the test size (Athanasiou et al., 2014). Fraser and Arcuri (2013) proposed an approach to reduce the number of generated tests by applying post-process minimisation.

6 Conclusions

In this paper, we have presented an approach to generate readable test data and implemented our approach in a tool called CTS. CTS can mine related constants from projects by analysing specific operators. Then, CTS can obtain heuristic information by aggregating and rating the mined related constants and use the heuristic information to generate integrated constraints for under-constrained variables or seed the test data generation process. Since most related constants are set by the developer according to certain specifications and are reasonable for the program under testing, the heuristic information is also reasonable. As a result, by using the heuristic information, CTS reduces the solution space of under-constrained variables and improves the efficiency of test data generation. Furthermore, due to the addition of heuristic information, the test data generated by CTS is smaller and more convenient for a human oracle.

Acknowledgements

This work was supported by the National Natural Science Foundation of China under Grant Nos. U1736110 and 61702044.

References

Afshan, S., McMinn, P. and Stevenson, M. (2013) 'Evolving readable string test inputs using a natural language model to reduce human oracle cost', in *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, Luxembourg, 18–22 March, pp.352–361.

Astronomy and Numerical Software Source Codes [online] http://www.moshier.net/ (accessed 16 January 2020).

Athanasiou, D., Nugroho, A., Visser, J. and Zaidman, A. (2014) 'Test code quality and its relation to issue handling performance', *IEEE Transactions on Software Engineering*, Vol. 40, No. 11, pp.1100–1125.

Beller, M., Gousios, G., Panichella, A. and Zaidman, A. (2015) 'When, how, and why developers (do not) test in their IDEs', in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, Bergamo, Italy, 30 August–4 September, pp.179–190.

Boonstoppel, P., Cadar, C. and Engler, D. (2008) 'RWset: attacking path explosion in constraint-based test generation', in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Budapest, Hungary, 29 March–6 April, pp.351–366.

Bozkurt, M. and Harman, M. (2011) 'Automatically generating realistic test input from web services', in *Proceedings of the 2011 IEEE 6th International Symposium on Service Oriented System*, Irvine, CA, USA, 12–14 December, pp.13–24.

Ceccato, M., Marchetto, A., Mariani, L., Nguyen, C.D. and Tonella, P. (2015) 'Do automatically generated test cases make debugging easier? An experimental assessment of debugging effectiveness and efficiency', *ACM Transactions Software Engineering Methodology*, Vol. 25 No. 1, pp.1–38.

Daka, E. and Fraser, G. (2014) 'A survey on unit testing practices and problems', in *Proceedings of the 2014 IEEE 25th International Symposium on Software Reliability Engineering*, Naples Italy, 3–6 November, pp.201–211.

Fraser, G. and Arcuri, A. (2013) 'Whole test suite generation', *IEEE Transactions on Software Engineering*, Vol. 39, No. 2, pp.276–291.

Fraser, G., Staats, M., McMinn, P., Arcuri, A. and Padberg, F. (2015) 'Does automated unit test generation really help software testers? A controlled empirical study', *ACM Transactions Software Engineering Methodology*, Vol. 24, No. 4, pp.1–49.

Gotlieb, A. (2015) 'Constraint-based testing: an emerging trend in software testing', *Advances in Computers*, Vol. 99, pp.67–101.

Li, B., Vendome, C., Linares-Vásquez, M., Poshyvanyk, D. and Kraft, N.A. (2016) 'Automatically documenting unit test cases', in *Proceedings of the 2016 IEEE International Conference on Software Testing, Verification and Validation*, Chicago, IL, USA, 11–15 April, pp.341–352.

McMinn, P., Shahbaz, M. and Stevenson, M. (2012) 'Search-based test input generation for string data types using the results of web queries', in *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, Montreal, QC, Canada, 17–21 April, pp.141–150.

McMinn, P., Stevenson, M. and Harman, M. (2010) 'Reducing qualitative human oracle costs associated with automatically generated test data', in *Proceedings of the First International Workshop on Software Test Output Validation*, Trento, Italy, pp.1–4.Mishra, P., Dutt, N., Krishnamurthy, N. and Abadir, M. (2005) 'A methodology for validation of microprocessors using symbolic simulation', *International Journal of Embedded Systems*, Vol. 1, Nos. 1–2, pp.14–22.

Neves, V.D.O., Delamaro, M.E. and Masiero, P.C. (2016) 'Combination and mutation strategies to support test data generation in the context of autonomous vehicles', *International Journal of Embedded Systems*, Vol. 8, Nos. 5/6, pp.464–482.

Offutt, A. and DeMilli, R. (1991) 'Constraint-based automatic test data generation', *IEEE Transactions on Software Engineering*, Vol. 17, No. 9, pp.900–910.

Palomba, F., Panichella, A., Zaidman, A., Oliveto, R. and De Lucia, A. (2016) 'Automatic test case generation: what if test code quality matters?', in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, Saarbrücken, Germany, 18–20 July, pp.130–141.

Pan, L., Wang, T., Qin, J. and Xiang, X. (2018) 'A dynamic test prioritisation based on DU-chain coverage for regression testing', *International Journal of Embedded Systems*, Vol. 10, No. 2, pp.113–119.

Panda, S. and Mohapatra, D.P. (2017) 'Hierarchical regression test case selection using slicing', *International Journal of Computational Science and Engineering*, Vol. 14 No. 2, pp.179–197.

Panichella, S., Panichella, A., Beller, M., Zaidman, A. and Gall, H.C. (2016) 'The impact of test case summaries on bug fixing performance: an empirical investigation', in *Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering*, Austin, TX, USA, 14–22 May, pp.547–558.

Shahbaz, M., McMinn, P. and Stevenson, M. (2012) 'Automated discovery of valid test strings from the web using dynamic regular expressions collation and natural language processing', in *Proceedings of the 2012 12th International Conference on Quality Software*, Xi'an, Shaanxi, China, 27–29 August, pp.79–88.

Shamshiri, S., Just, R., Rojas, J.M., Fraser, G., McMinn, P. and Arcuri, A. (2015) 'Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges (T)', in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering*, Lincoln, NE, USA, 9–13 November, pp.201–211.

Vardy, A. and Moller, R. (2005) 'Biologically plausible visual homing methods based on optical flow techniques', *Connection Science*, Vol. 17, Nos. 1–2, pp.47–89.

Xing, Y., Gong, Y., Wang, Y. and Zhang, X. (2014) 'Branch and bound framework for automatic test case generation', *SCIENTIA SINICA Informationis*, Vol. 44 No. 10, pp.1345–1360.