International Journal of Software Engineering
and Knowledge Engineering
Vol. 29, No. 9 (2019) 1279–1312

DOI: 10.1142/S0218194019500402

World Scientific
www.worldscientific.com

Improvements of Directed Automated Random Testing in Test Data Generation for C++ Projects

Duc-Anh Nguyen*, Tran Nguyen Huong†, Hieu Vo Dinh‡
and Pham Ngoc Hung§

VNU University of Engineering and Technology
Hanoi, Vietnam
**nguyenducanh@vnu.edu.vn*
†*17028005@vnu.edu.vn*
‡*hieuvd@vnu.edu.vn*
§*hungpn@vnu.edu.vn*

Received 21 July 2018
Revised 26 August 2018
Accepted 8 February 2019

This paper improves the breadth-first search strategy in directed automated random testing (DART) to generate a fewer number of test data while gaining higher branch coverage, namely Static DART or SDART for short. In addition, the paper extends the test data compilation mechanism in DART, which currently only supports the projects written in C, to generate test data for C++ projects. The main idea of SDART is when it is less likely to increase code coverage with the current path selection strategies, the static test data generation will be applied with the expectation that more branches are covered earlier. Furthermore, in order to extend the test data compilation of DART for C++ context, the paper suggests a general test driver technique for C++ which supports various types of parameters including basic types, arrays, pointers, and derived types. Currently, an experimental tool has been implemented based on the proposal in order to demonstrate its efficacy in practice. The results have shown that SDART achieves higher branch coverage with a fewer number of test data in comparison with that of DART in practice.

Keywords: Directed automated random testing; concolic testing; test data compilation; test data generation; control flow graph; C++; SMT-Solver.

1. Introduction

Unit testing has been considered an important phase to ensure the high quality of software, especially for the system software written in C++ due to the painstaking requirements of quality. Two well-known approaches for unit testing are *black-box testing* and *white-box testing* [22]. Black-box testing only focuses on the correctness of input and output without considerations about its source code. In contrast,

§Corresponding author.

white-box testing tries to inspect the quality of source code by analyzing it. This approach allows detecting potential errors in software that cannot be found by black-box testing. However, the cost for evaluating the quality of software is quite expensive, especially in large-scale software. The need for automated unit testing is becoming more and more urgent and unavoidable to reduce the budget of the testing phase. Up to the present, there are two major directions of automated test data generation known as *static testing* and *dynamic symbolic execution* (DSE) [3]. The idea of the former is to generate test data automatically by applying source code analysis techniques. Although this direction seems to be effective, it faces several issues in practice. The main reason is that the source code analysis requires a large amount of effort to deal with various syntaxes such as API usage, lambda, etc. The latter can be divided into two major methods including execution generated testing (EGT) and concolic testing. EGT aims to detect possible potential bugs in C++ projects by executing it symbolically in up to thousands of separate forks [5, 6, 14]. In contrast, concolic testing, which is first implemented in directed automated random testing (DART) [10], aims to produce a series of test data maximizing a specific code coverage criterion with the least number of test data.

In DART, given a function under test, initial test data are generated randomly based on the type of passing variables. These initial test data are then passed into an instrumented program to execute it on the runtime environment. During this test data execution, DART collects path constraints, denoted by PC, at each decision until the testing program raises an error or terminates. In the first case, a bug is reported and DART starts generating another test data randomly again. Otherwise, DART negates the current path constraints PC in the way that the solution of the negated path constraints tends to visit the unvisited branches when executing it on the testing function. Although DART demonstrates its effectiveness in practice, this method still remains an issue related to the number of test data. In addition, DART currently provides a fast test data compilation mechanism to reduce the computational cost of test data generation, but only supports C projects.

Regarding the number of test data in concolic testing, it should be minimized to facilitate the testing management process with the smaller number of iterations. In order to achieve this objective, DART tries to lower the number of iterations as many as possible. Generally, the process of the next test data generation in DART includes four main steps: (i) generating path constraints from the current test path, (ii) negating these path constraints PC to get $\neg PC$, (iii) generating the next test data by solving $\neg PC$, and (iv) executing the next test data to get the next test path. The solution to reduce the number of iterations depends mainly on step *ii*, where a constraint in PC is negated to generate the next test data so as to go through unvisited branches. However, in fact, the negated path constraints $\neg PC$ could not ensure completely that its solution will pass through unvisited branches due to several reasons. One main reason is that there might exist many candidate constraints to negate based on the selected path selection strategy [9–11, 13]

(e.g. breadth-first search (BFS), depth-first search (DFS), etc.) and how to choose the best one is still a challenging problem. In the worst case, it might take a large number of iterations to achieve high code coverage due to the selected strategies. Another reason is related to the capacity of symbolic execution. Specifically, given a test path, a set of path constraints will be generated by applying symbolic execution [2, 3]. Each path constraint in this set is corresponding to a condition on the given test path. These path constraints are then negated and solved to obtain the next test data [8, 11, 17–19]. However, in the case where the implementation of SE does not support enough cases (e.g. pointers, two-dimensional arrays), the next test data will not go through the new visited branches as expected. As a result, the computational cost of test data generation might increase significantly.

Currently, the test data compilation suggested in DART only applies to basic types, pointer, struct, and array in C projects. Therefore, when generating test data for C++ projects, this compilation mechanism needs to be improved to reduce the computational cost of test data generation. According to DART, when new test data are discovered, then they are executed on runtime environment to collect visited instructions by a general test driver. The main idea of the test data compilation in DART is to store the generated test data in a unique external file, and then a test driver analyzes this file to load the values of test data when executing. One big advantage of this strategy is that the general test driver only needs to compile once to create an executable file. Because of one-time compilation, the total computational cost of test data compilation, in particular, and test data generation in general are reduced significantly, especially in the projects having a large number of test data.

Therefore, this paper proposes two techniques to deal with the mentioned limitations. First, in order to reduce the number of test data, the paper improves the BFS strategy proposed in DART by combining this strategy with a static test data generation strategy, namely Static DART, or SDART for short. Specifically, when it is less likely to increase code coverage with the current path selection strategy, the static test data generation strategy will be selected instead. In this static analysis strategy, SDART will generate a list of partial test paths which go through unvisited branches, then try to generate test data traversing these partial test paths. In other words, by combining the existing path selection strategies with the static test data generation strategy, SDART expects that more newly visited branches will be detected earlier than keeping the current strategies. Second, the paper extends the idea of the test data compilation for C projects to deal with C++ projects by using a C++ general test driver. In essence, the idea of the general C++ test driver is similar to the test driver used in the test data compilation proposed in DART. However, the C++ general test driver is presented in a more general representation by using templates. By using templates, the C++ general test driver is more flexible and expandable to support various data types.

The structure of the paper is organized as follows. Several outstanding related works are discussed in detail in Sec. 2 to provide the overview of the test

data generation. Section 3 presents the background of DART. After that, Sec. 4 provides the overview of the proposed method and the description of source code preprocessing phase. Next, the details of the second phase called test data generation are described in Sec. 5. Section 6 shows an implemented tool named ICFT4Cpp based on the proposed method to demonstrate its effectiveness in practice. Finally, the conclusion of the paper is given in Sec. 7.

2. Related Works

Many works have been proposed for enhancing test data generation phase by several research groups. Focus on the most outstanding works only, there are seven effective improvements of test data generation for C/C++ projects including test data compilation [10, 13], compositional testing [21], symbolic execution [8, 11, 17–19], constraints optimization [6, 8, 11, 14, 15], parallel SMT-Solvers [16], path selection strategies [8–11, 13], and initial test data generation [23].

Godefroid *et al.* extended DART for the purpose of compositional testing by introducing an algorithm, namely SMART (*Systematic Modular Automated Random Testing*) [21]. SMART tests functions in isolation, then encoding test results as function summaries expressed using input preconditions and output post-conditions. After that, these summaries are used for testing higher-level functions. Currently, our method does not focus on testing compositional functions.

The computational cost of test data generation can be reduced significantly in the test data compilation step. Both DART [10] and CREST [13] applied the same technique to accelerate the test data compilation step. CREST is known as an open-source prototype test generation tool for C. The main idea is that all of the generated test data are stored in an external file, and a unique test driver reads this file to collect the values during execution. It means that the compilation process of general test driver takes place once for all of the produced test data. However, CREST proposal is only applied for basic types rather than for derived types which are used widely on C++ projects. In addition, the method proposed in DART limits on C projects. Therefore, our proposed method is developed based on the original idea of CREST and DART to deal with not only basic types but also derived types (i.e. *class*, *struct*) on C++ projects.

Because the constraints generated from a test path may be complicated and lengthy, SMT-Solvers may take a long time to solve these constraints. To reduce the cost of solving these constraints, these constraints will be optimized before passing into SMT-Solvers. There are three main types of constraints optimization. The first optimization named *incremental solving* technique is used in CUTE [11], EXE [14], KLEE [6], and CAUT [8]. The main idea is that only the constraints related to the last negated condition are solved rather than all of the original constraints. The second optimization, which is called *cache-based unsatisfiability check*, is implemented in EXE [14] and KLEE [6]. In this optimization, all of the previous constraints are cached for the next solving constraints. A subset of constraints

having no solution means that the current constraints are unsatisfiable. Otherwise, the current constraints may have a solution if there exists a subset of constraints having a solution. Third, the constraint optimization in CUTE [11] and CAUT [8, 12] tries to check whether the last condition is a negation of any previous constraints. If it is, SMT-solvers do not solve these constraints because it is always impossible. Another constraint optimization removes evident subconstraints from the original constraints so as to reduce the complexity of these constraints [6, 11]. Recently, in [15], Cadar *et al.* proposed a new constraint optimization for array case. Their paper introduced three transformations called index-based transformation, value-based transformation, and interplay of transformations to decrease the complexity of array constraints.

Another strategy to improve the computational cost of test data generation is to combine several SMT-Solvers such as Z3 [4], SMT-Interpol, STP [6, 14], etc. In fact, each of SMT-Solvers only deals effectively with some types of constraints. In [16], Cadar *et al.* proposed some ideas of SMT-Solvers combination for finding a solution as fast as possible called *parallel portfolio solver*. Specifically, their portfolio solver can be deployed at different levels of granularity. In the simplest option called *program level*, multiple variants of symbolic execution are run at the same. Each variant uses a distinct SMT-Solver to solve constraints. In the second option, the test data generation process is performed in a unique machine, but at *query level*. Given a query, the machine detects the most suitable SMT-Solver for the current query because different SMT-Solvers may perform better on different queries. Another idea is that instead of using different SMT-Solvers, the machine performs with different versions of a unique SMT-Solver.

Several path strategies have been proposed to reduce the number of test data while trying to maximize the code coverage in concolic testing. Initially, DART [10] proposed DFS strategy negating the last condition to find the next test data. This strategy actually increases code coverage; however, it may lead to the run-forever problem, especially in the functions containing loops. Later, the run-forever problem can be solved by adding the number of limit iterations for loops in PathCrawler [9] and CUTE [11]. CREST [13] uses the Dijkstra algorithm to find the shortest path from the visited statements/branches to the unvisited instructions. Rather than using Dijkstra algorithm, CAUT [8, 12] tries to find the best path from visited instructions to the uncovered block of instructions.

Nguyen *et al.* suggested that the initial test data should be generated by static analysis due to the problem of random test data generation [23]. The idea is to construct a control flow graph, then generate initial test data based on this graph by using the symbolic execution. Because the constraints between variables are detected during static analysis, the probability of runtime error executions caused by the initial test data tends to reduce in comparison with that of the random technique. For example, by using static analysis, a pointer must be always allocated in memory. Therefore, the value of this pointer in the initial test data should not be assigned to *NULL*.

3. Directed Automated Random Testing

DART can deal with the disadvantages of static testing requiring a large amount of effort in symbolic execution. DART defines that a sequence of input addresses, denoted by $\mathbf{M}_0$, is the addresses of the parameters of a function *fn*. There are many input vectors corresponding to a sequence $\mathbf{M}_0$ in which an input vector represents the value of a parameter. In concolic testing, the initial input vector is generated at random, e.g. pointers are randomly initialized with either the value *NULL* or a new memory with the equal probability.

Definition 1 (Input vector). An input vector of a function *fn* is defined as follows:

$$\mathbf{I} = (t_0, t_1, \ldots, t_{n-1}),$$

where

- n is the number of parameters (i.e. arguments, external variables) used in *fn* and
- $t_i = (nameVar, valueVar | valueVar \in \{NULL, !NULL, number, character, string\})$ $(0 <= i <= n-1)$ is a parameter, where *nameVar* is the name of a parameter and *valueVar* is the value of the parameter having the name *nameVar*.

Here, the type of t_i is an element of an array or of a pointer, derived type, or basic type. The value of t_i is a NULL value, a not NULL value, a number, a character, or a string.

In concolic testing, the next input vector will be generated based on the results of the previous input vector executions. However, an input vector must be executed on an instrumented function fn', where fn' is a modification version of the function *fn* after adding some marked statements. Specifically, before input vector execution, each statement in the function *fn* will be added a marked statement to detect whether this statement is executed or not under an input vector.

When an input vector $\mathbf{I}$ is executed on an instrumented function fn', a list of visited branches are recorded after an input vector execution. These branches form a *program execution path*. There are two cases happened when executing fn' under an input vector $\mathbf{I}$ including *abort* and *halt*. Here, signal *abort* means that there arise errors while signal *halt* indicates that the function *fn* succeeds.

Definition 2 (Program execution path). A program execution path of a function *fn*, generated when executing an input vector $\mathbf{I}$, is defined as a sequence:

$$TP = (S|C)^+(abort|halt),$$

where S and C present a statement and a condition of the function *fn*, respectively.

During the execution of the instrumented function fn' under an vector input $\mathbf{I}$, DART uses a symbolic map to store the symbolic value of variables.

Definition 3 (Symbolic map). Given a program execution path of a function *fn*, its symbolic map, denoted by S, is a mapping from variables to its corresponding

symbolic values and defined as follows:

$$S = \{(var, symbolic\_value)^+\},$$

where *var* presents a variable; *symbolic_value* is the symbolic value of the variable *var*. The symbolic value of *var* is a concrete value (e.g. a constant, a string) or an expression. When *var* is updated, its symbolic value to a new value named *new_symbolic_value*, S is updated by using operator $\cup$ as follows:

$$S = S \cup \{(var, new\_symbolic\_value)\}.$$

When an input vector $\mathbf{I}$ is executed, DART also performs symbolic execution to update the state of memory model M. In essence, memory model M takes responsibility for storing the current address of variables. A variable is a parameter or a local variable. After executing a statement or a condition changing the address of variables, the state of memory model M is updated.

Definition 4 (Memory model). Given a program execution path of a function *fn*, its memory model M is a mapping from memory addresses to variables, and defined as follows:

$$M = \{(addr, var)^+\},$$

where *addr* is the address of a variable; *var* is the name of the variable having the address *addr* in memory model M. When *var* is updated to a new address named *new_addr*, M is updated by using operator $\cup$ as follows:

$$M = M \cup \{(new\_addr, var)\}.$$

In order to distinguish the difference between symbolic map S and memory model M, let see via the following example. Assume that DART visits the statement $z = x + y + 2$, where both x, z are local variables; and y is a parameter. The value of x and z is set to 1 and 0 previously, respectively. Before analyzing the assignment of z, the state of memory model M and symbolic map S are as follows:

$$M = \{(addr(y), y), (addr(x), x), (addr(z), z), (addr(p), p)\},$$
$$S = \{(x, 1), (y, Y), (\mathbf{z}, \mathbf{0}), (p[0], 1), (p[1], 2)\},$$

where $addr(x)$, $addr(y)$, $addr(z)$, $addr(p)$ are the address of the variable x, y, z, and p, respectively; Y is the initial symbolic value of the parameter y; p is a one-level pointer of size 2.

After analyzing this statement, the state of memory model M and symbolic map S are updated as follows:

$$M = \{(addr(y), y), (addr(x), x), (addr(z), z), (addr(p), p)\},$$
$$S = \{(x, 1), (y, Y), (\mathbf{z}, \mathbf{1} + \mathbf{Y} + \mathbf{2}), (p[0], 1), (p[1], 2)\},$$

where $1 + Y + 2$ is the new symbolic value of z.

Now, the next visited statement is a condition $p! = NULL$. In order to evaluate the boolean value of this statement, we need to check the address of p stored in memory model M whether p is allocated or not. After checking, we found out that pointer p is allocated before. Therefore, the next condition returns *false*.

Path constraints are collected during DSE under a vector $\mathbf{I_i}$, then used to find the next input vector $\mathbf{I_{i+1}}$ by negating the current path constraints. Generally, the number of path constraints is equivalent to the number of conditions on the *program execution path*.

Definition 5 (Path constraints). Given a program execution path of a function fn, its corresponding path constraints are defined as a logic expression, defined as follows:

$$PC = pc_0 \wedge pc_2 \wedge \cdots \wedge pc_{n-1},$$

where n is the number of conditions on TP; pc_i is a constraint $(0 <= i <= n-1)$; pc_0 and pc_{n-1} are the path constraints corresponding to the first condition and the last condition in TP, respectively.

After glancing at the fundamental definitions used in DART, we move to the idea of DART presented in Algorithm 1. Given a function named fn, DART aims to generate a series of test data satisfying branch coverage, e.g. all branches in the function fn are visited. However, the test data generation will perform on the instrumented function of fn' other than the original function fn. The main reason is that fn' has the same behavior as fn. The only difference is that fn' is added marked statements to record the visited statements and visited branches when executing a test data. The process of function instrumentation will be discussed in detail in Sec. 4.2. The parameter $DEPTH$ is used to specify the number of times the top-level function is to be called iteratively in a single run. For each value of *depth*, the initial test data is generated at random.

Algorithm 1. The General Idea of DART

Input: fn': the instrumented function of a function fn, $DEPTH$: the depth of test data generation

Output: a series of test data

```
1: P = get arguments and external variables in fn'
2: for int depth = 0; depth <DEPTH; DEPTH++ do
3:    I_0 = random_initialization(P)
4:    while compute_coverage(fn') <100% do
5:       I_{i+1} = instrumented_program(fn', I_i)
6:       if I_{i+1} does not exist then
7:          I_{i+1} = random_initialization(P)
8:       end if
9:    end while
10: end for
```

Algorithm 2 illustrates how DART can generate the next input vector from the current input vector. The input includes the instrumented function fn' and the current input vector $\mathbf{I_i}$. The objective of each iteration presented in this algorithm is to find the next input vector $\mathbf{I_{i+1}}$, or a bug. Initially, memory model M is initialized from the input vector $\mathbf{I_i}$ (line 1). Along with step, symbolic map S is also initialized to store the symbolic value of parameters (line 2). The current statement is detected by using the function *statement_at*(*counter, M*), where *counter* is the index of this statement in the instrumented function fn' (line 4). The value of *counter* is changed when DART moves to a new statement. At this step, the statement s is executed to get its state of execution (i.e. **abort**, or **halt**). In the case there is no error, based on the type of statement s, there are two cases as follows.

Algorithm 2. Instrumented_program (in *DART*)

Input: fn': the instrumented function of fn, $\mathbf{I_i}$: the current input vector
Output: $\mathbf{I_{i+1}}$: the next input vector

```
1: Initialize a memory model M from I_i
2: Set up the state of a symbolic map S
3: counter := 0
4: s = statement_at(counter, M)
5: while s ∉ {abort, halt} do
6:    if s is (m ← e) then
7:       Update memory model M
8:       Update value of variable m in S
9:       counter++
10:   else if s is a condition C then
11:      b = evaluate_concrete(C, M)
12:      pc = evaluate_symbolic(C, M, S)
13:      if b is true then
14:         PC.add(pc)
15:         Update visited branches
16:         counter++
17:      else
18:         PC.add(¬pc)
19:         Update visited branches
20:         Set counter to the false branch position
21:      end if
22:   end if
23:   s = statement_at(counter, M)
24: end while
25: if s causes abort then
26:    return a bug
27: else
28:    ¬PC = Negate PC by applying a path selection strategy
29:    I_{i+1} = Solve(¬PC)
30: end if
```

In the first case, if the statement s is an assignment, the value of variable m will be updated in memory model M. In addition, the symbolic value of variable m is updated in symbolic map S; and the value of variable *counter* increases by one (lines 6–9).

In the second case, if the statement s is corresponding to a condition C, DART will evaluate the value of C by replacing the variables used in C with its corresponding values (line 11). Simultaneously, a new constraint pc is created through the process of evaluating symbolically this constraint (line 12). If *evaluate_concrete* returns *true*, it means that DART will visit the *true* branch under the vector $\mathbf{I_i}$. Therefore, pc is added to PC and the visited branches set is updated (lines 13–16). Otherwise, PC will add the negation of the condition C, then the value of visited branches set and of *counter* are updated at the same time.

Next, the next statement is obtained and this process proceeds repeatably until the execution returns an error (**abort**) or a success signal (**halt**) (lines 20–23). If a signal **halt** returns, DART tries to negate the current path constraints PC by applying BFS, DFS, or another strategy. After that, DART calls an SMT-Solver, e.g. lp_solve, to get a new input vector $\mathbf{I_{i+1}}$.

In this paper, the term *test data* is used rather than *input vector* for consistency. Similarly, instead of using *program execution path*, we refer to an equivalent term, namely *test path*.

4. The Proposed Overview and Source Code Preprocessing

The overview of the proposed method includes two major phases named *source code preprocessing phase* and *test data generation phase* (shown in Fig. 1). Specifically, the input of the first phase includes a C++ project and the specification of the operating system which the project run on. Initially, the source code preprocessing phase takes these inputs to remove all preprocessor directives existing in the given project. Next, the corresponding structure tree of the modified project is constructed. Simultaneously, all functions in the testing project are inserted extra codes in order to collect the visited statements and visited branches when these functions are called.

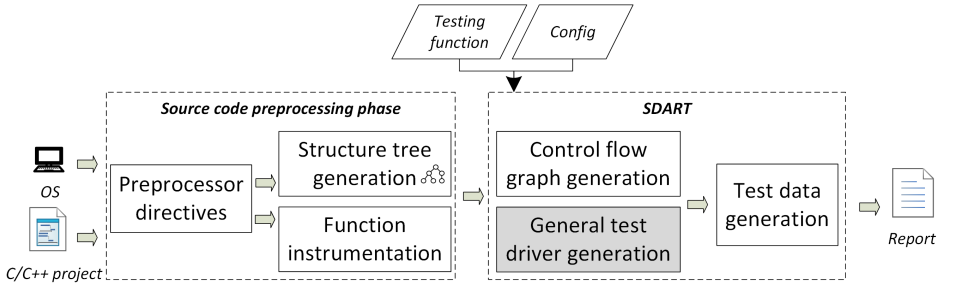

Fig. 1. The overview of the proposed method.

These functions are instrumented in such a way that visited statements and visited branches will be printed to an external test data file when executing the testing function. Later, the content of this external test data file is analyzed to get the corresponding test path.

Given a function *fn* and its configuration, the second phase takes responsibilities for generating a series of test data. The configuration of the testing function *fn* includes the bounds of variable types (i.e. number, character), the maximum iteration of loops, and the maximum size of arrays. First, both the control flow graph and the general test driver of the testing function *fn* are created at the same time. In the next step, a series of test data are produced, and then unit test files are represented in the form of Google Test and MS. Excel files are created automatically to facilitate testing management steps.

In the second phase, when a number of continuous test data do not increase coverage by using the BFS strategy, the static test data generation will be performed instead. In this situation, all possible partial test paths traversing through unvisited branches will be generated automatically. After that, a series of test data are produced from these partial test paths with the expectation that newly uncovered branches will be traversed as soon as possible.

Also in the second phase, SDART extends the test data compilation mechanism proposed in DART to deal with C++ projects by using a C++ general test driver. This test driver provides the ability to deal with various data types. An external data file, which takes responsibility for storing test data, is unique during test data generation. Whenever new test data are created, the content of the external test data file is updated. The executable test driver will load the value of test data storing in the external test data file to initialize variables dynamically.

4.1. *Structure tree generation*

The first phase of source code preprocessing aims to construct an intermediate representation of the given C++ project. The structure tree is based on the well-known composite pattern [7] as the following definition.

Definition 6 (Node). A node of a structure tree S, denoted by nd, is defined as follows:

$$nd = (X, np, D),$$

where

- X is the subcomponents of the node nd (e.g. files in a folder),
- np presents the parent of the node nd (e.g. a folder containing a list of files), and
- D represents the logic dependencies of the node nd with the other nodes in the structure tree S. The type of the node nd is folder, file (e.g. source code file, header file); or logic element such as derived type, method, attribute, etc.

Definition 7 (Structure tree). Given a C++ project, its corresponding structure tree, denoted by S, is defined as follows:

$$S = (V_k, E),$$

where

- k is the number of nodes in the structure tree,
- $V_k = \{nd_0, nd_1, \ldots, nd_{k-1}\}$ is a list of nodes, and
- $E = \{(nd_i, nd_j)^*\} \subset V_k \times V_k$ presents a list of edges $(0 <= i, j <= k-1, nd_i \in V_k, nd_j \in V_k)$. An edge (nd_i, nd_j) means that node nd_j is a subcomponent of node nd_i.

Consider a structure tree $S = (V_k, E)$, among two nodes nd and nd', where $(nd, nd') \notin E$, there may exist several logic dependencies starting at nd and finishing at nd'. Each logic dependency represents a type of relationship between two nodes nd and nd'. For example, Table 1 illustrates several typical types of logic dependency in a structure tree. Specifically, the first dependency describes a relationship between a method and an attribute. In this logic dependency, the method is considered as a getter of the mentioned attribute in a class. In the case if a class extends another class, there is a logic dependency starting at *derived class* and finishing at *base clas.*

Table 1. Several logic dependencies of class elements in structure tree.

Start dependency	End dependency	Description
Method (getter)	Attribute	The getter of an attribute, e.g. getter getAge() gets the value of age
Method (setter)	Attribute	The setter of an attribute, e.g. setAge(int) sets the value of age
The declaration of a method in class	The definition of method	Method is defined outside its class
Derived class	Base class	An inheritance between two classes, e.g. class Student: public People

Figure 2 gives an example of the structure tree corresponding to a portion of *Algorithm* project. The function *NaivePatternSearch* mentioned above is declared inside namespace *Algorithm* in header *Search.h* and defined in *Search.cpp*. Therefore, there is an arrow from the definition of this method to its declaration. In brief, the structure tree provides an overview of a C++ project. It is especially appropriate for traversing the given project to seek information. For example, consider *Sort.cpp*, whenever we need to get its headers, we only traverse the children of the node *Sort. cpp* with logic dependencies labeled *include.*

4.2. *Function instrumentation*

In the preprocessing phase, not only structure tree generation but also function instrumentation is performed. Specifically, the instrumentation stage is an

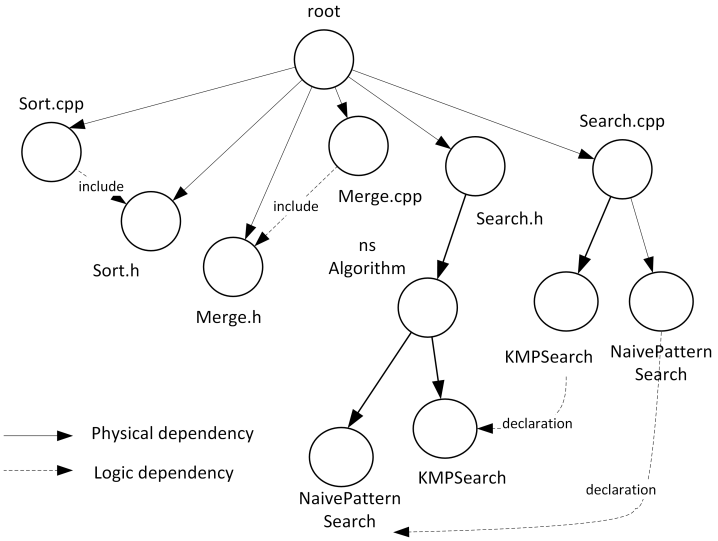

Fig. 2. A structure tree portion of the project *Algorithm*.

automated transformation of functions so as to print out test path after executing this function. In general, there are many techniques to add addition codes to these functions. In the traditional technique, the functions are rewritten by applying regular expressions. Nonetheless, because statements may be represented in various syntaxes, it is challenging to define it through regular expressions exactly. This technique therefore seems to be less effective in practice.

Instead of applying the same manner, the modern technique transforms these functions into corresponding Abstract Syntax Trees (ASTs). Given a function under test, its corresponding AST is an intermediate representation in which each node of this AST denotes an element of source code (e.g. if-block, assignment, *return*/*break*/*continue* statement, etc.). The element storing in a node can be divided into smaller elements (e.g. an if-block can be decomposed into two smaller elements referred to its body and its decision). In this case, the nodes corresponding to the smaller elements are the children of the node containing the broken element. If an element is an assignment, a decision of a control block, a declaration, or a control statement (e.g. continue, break), this element cannot be broken. In this situation, these types of element are the leaves of AST.

This technique has been used in the front-end phase of the modern compilers, e.g. Clang,[a] GCC,[b] etc. In the proposed method, these functions are instrumented based on its AST so that the proposed instrumentation is more accurate than the traditional technique. In typical C++ functions, there exist single declaration/assignment

[a] https://clang.llvm.org/.
[b] https://gcc.gnu.org/.

statements, return/break/continue/throw, control blocks including *while…do*, *do… while*, *for*, *if…else*, etc. The instrumented rules of these statements are listed in Table 2. In this table, the notation ⟨expression⟩ means the content of expression. The role of the function *mark*(*str*) is to print out the string *str* to a specific file.

Table 2. List of instrumentation rules.

Type of block A	Instrumented block
assignment, declaration, throw/break/continue/ return, }, {	mark(“⟨A⟩”); A
while(⟨condition⟩)...do{...}	while (mark(“⟨condition⟩”) && ⟨condition⟩)do{...}
do{...}while(⟨condition⟩)	do{} while (mark(“⟨condition⟩”) && ⟨condition⟩)
if (⟨condition1⟩){...}else if (⟨condition2⟩){...}else{...}	if (mark(“⟨condition1⟩”) && ⟨condition1⟩){...}elseif (mark(“⟨condition2⟩”) && ⟨condition2⟩){...}else{...}
for(init, condition, increment){...}	for(mark(“⟨init⟩”) && init, mark(“⟨condition⟩”) && condition, mark(“⟨increment⟩”) && increment){...}
try{...}catch(⟨exception1⟩){...}	mark(“try”);try{...}
catch(⟨exception2⟩){...}	catch(⟨exception1⟩){mark(“⟨exception1⟩”);...} catch(⟨exception2⟩) {mark(“⟨exception2⟩”);...}

Based on these rules, an example of the instrumented function of *Algorithm::Utils::Fibonacci* is illustrated in Listing 1. Specifically, before the condition, $n == 0$ is a corresponding marked statement $mark(“n == 0”)$ to print out the content of this condition whenever it is executed. Here, the condition $n == 0$ and its marked statement $mark(“n == 0”)$ put the condition of the block *if*.

Listing 1. The instrumented function of *Algorithm::Utils::Fibonacci*.

```
namespace Utils{
    int Fibonacci(int n){
        if (mark(``n == 0'') && n == 0){
            mark(``return 0'');
            return 0;
        }else if (mark(``n == 1'') && n == 1){
            mark(``return 1;'');
            return 1;
        }else{
            mark(``return (Algorithm::Utils::Fibonacci(n-1) +
                           Algorithm::Utils::Fibonacci(n-2))'');
            return (Algorithm::Utils::Fibonacci(n-1) +
                    Algorithm::Utils::Fibonacci(n-2));
        }
    }
}
```

5. An Improvement of DART

Static DART, or SDART for short, extends DART to reduce the number of test data with the smaller number of iterations by improving the BFS strategy introduced in DART [10]. SDART is presented in Algorithm 3, where the notation $(*)$ at the end of a statement implies that this statement exists in the original algorithm proposed in DART.

Algorithm 3. SDART

Input: fn' instrumented function of a function fn **(*)**, $DEPTH$: depth of test data generation **(*)**, $THRESHOLD$: the threshold to switch to the static test data generation

Output: a series of test data

1: P = get arguments and external variables in fn' **(*)**
2: $Cpp\_testdriver$ = Cpp-general-test-driver-generation(**I**)
3: **for** int $depth = 0$; $depth < DEPTH$; $DEPTH{+}{+}$ **(*) do**
4: **I**$_0$ = random_initialization(P) **(*)**
5: $notIncreasingCoverageCount = 0$;
6: **while** compute_coverage(*fn'*) <100% **(*) do**
7: **if** the coverage of fn does not change **then**
8: $notIncreasingCoverageCount{+}{+}$;
9: **else**
10: $notIncreasingCoverageCount = 0$;
11: **end if**
12: **if** $notIncreasingCoverageCount == THRESHOLD$ **then**
13: break
14: **else**
15: **I**$_{i+1}$ **= instrumented_program(**fn'**,** **I**$_i$**) (*)**
16: **if** **I**$_{i+1}$ does not exist **(*) then**
17: **I**$_{i+1}$ = random_initialization(P) **(*)**
18: **end if**
19: **end if**
20: **end while**
21: $possibleTestpaths$ = generate partial test paths containing unvisited branches
22: **for** $testpath$: $possibleTestpaths$ **do**
23: PC = symbolic-execution($testpath$)
24: **if** PC does not exist before **then**
25: $SMT - LIB$ = Transform PC into SMT-Lib format
26: **I** = SMT-Solver($SMT - LIB$)
27: $testpath$ = Execute $Cpp\_testdriver$
28: compute_coverage(*fn'*)
29: **end if**
30: **end for**
31: **end for**

Generally, when there occur signals showing that they are less likely to increase code coverage, the static test data generation will be chosen immediately. Specifically, the value *THRESHOLD* is initialized by users. When there exists *THRESHOLD* continuous test data which do not increase code coverage (lines 7–13), SDART will generate partial test paths traversing unvisited branches (line 21). After that, each test path performs symbolic execution so as to construct corresponding path constraints (line 23). These path constraints *PC* are checked whether they are generated before or not (line 24). If it is not, *PC* is converted into input of SMT-Solvers, then solved by using an SMT-Solver such as Z3 to obtain new test data **I** (lines 25–26). Next, the newly generated test data **I** are passed to a general test driver to execute to get its corresponding test path (line 27). After that, from the collect test path, the coverage of *fn* is then updated (line 28).

5.1. *Path constraint generation*

This section describes the process of path constraint generation from a test path by applying symbolic execution technique. This section includes two parts. First, the paper presents the detail of symbolic execution on a test path. Next, the process of simplification, denoted by $rewrite()$ and mentioned in the first step, will be described in more detail in the second part.

Both Algorithms 4 and 5 use notation μ to denote binary expression. A binary expression, denoted by $\mu(e_1, e_2, op)$, consists of one operator op, one left operand e_1, and one right operand e_2. An assignment is a binary expression, where $op \in \{=\}$. A condition is a binary expression where $op \in \{>, >=, <=, <, ==, !=\}$, or a unary expression. $M(v)$ is used to get the address of variable v, where M is memory model. $S(v)$ returns the symbolic value of variable v, where S is symbolic map. Notation $\equiv$ means "*is type of*" or "*is*".

5.1.1. *Symbolic execution*

The process of symbolic execution is presented in Algorithm 4. Our symbolic execution uses a memory model and a symbolic map presented in DART, namely M and S, respectively. The role of memory model M is to store the addresses of the used variables during symbolic execution. Symbolic map S takes responsibility for storing the symbolic value of variables. The algorithm terminates when one of these two cases happens: (1) the boolean value of a condition is *false* or (2) all of the statements on the test path are analyzed. In the first case, whenever Algorithm 4 finds out that path condition *PC* has no solution, it stops immediately to reduce the cost of symbolic execution.

Procedure $type\_of(v)$ returns the type of variable v. Procedure $addr(v)$ is used to obtain the address of variable v stored in memory model M. Procedure $addr(v)^k$ will get the beginning address of the block in $addr(v)$ after k cells ($k \in \mathbb{Z}$). In order to specify the number of elements in the block which a pointer p points to, the paper uses the procedure $sizeof(p)$.

Algorithm 4. Symbolic Execution on C++

Input: *path*: a test path collected from executing the instrumented function fn'
Output: PC: the path constraints of the test path *path*

1: Symbolic map $S = \{(parameter, initial\_symbolic\_value)^*\}$
2: Memory model $M = \{(initial\_address\_of\_parameter, parameter)^*\}$
3: **for all** statement *stm*: *path* **do**
4: *stm* = Rewrite(*stm*, M, S)
5: **if** $stm \equiv \mu(e_1, e_2, op | op \in \{>=, >, <=, <, ==, !=\}) \| stm \equiv$ a boolean variable **then**
6: **if** $stm \equiv false$ **then**
7: $PC = \{\}$
8: return PC
9: **else**
10: PC.add(*stm*)
11: **end if**
12: **else if** $stm \equiv$ declaration of variable v **then**
13: $M = M \cup \{(addr(v), v)\}$
14: **if** $type\_of(v) \in \{number, character\}$ **then**
15: $S = S \cup \{(v, 0)\}$
16: **else if** $type\_of(v) \in \{pointer\}$ **then**
17: $S = S \cup \{(v, NULL)\}$
18: **else if** $type\_of(v) \in \{string\}$ **then**
19: $S = S \cup \{(v, \text{“”})\}$
20: **end if**
21: $v.setScope(scope)$
22: **else if** $stm \equiv \mu(e_1, e_2, =)$ **then**
23: **if** $e_1 \equiv$ a pointer **then**
24: **if** $e_2 \equiv$ an allocation statement (size = n) **then**
25: $M = M \cup \{(addr(e_1), e_1)\}$
26: $S = S \cup \bigcup_{i=0}^{n-1} \{(e_i, 0)\}$
27: PC.add(n >= 0)
28: **else if** $e_2 \equiv NULL$ **then**
29: $M = M \cup \{(NULL, e_1)\}$
30: $S = S \cup \{(e_1, NULL)\}$
31: **else if** $e_2 \equiv$ a pointer p +/- k **then**
32: $M = M \cup \{(M(p)^k, e_1)\}$
33: $S = S \cup \bigcup_{i=0}^{sizeof(p)-k-1} \{(e_1(i), p_{i+k})\}$
34: **end if**
35: **else if** $type\_of(e_1) \in \{number, character\}$ **then**
36: $S = S \cup \{(e_1, e_2)\}$
37: **end if**
38: **else if** $stm \equiv$ scope **then**
39: **if** $stm \equiv$ ‘{’ **then**
40: $scope++$;
41: **else if** $stm \equiv$ ‘}’ **then**
42: $M.removeVariablesAtScope(scope)$
43: $S.removeVariablesAtScope(scope)$
44: $scope--$;
45: **end if**
46: **end if**
47: **end for**
48: return PC

Given a test path collected from executing the instrumented function fn', a memory model M is created by adding parameters and its initial addresses (line 1). These parameters are added to a symbolic map S with its default symbolic values if it is not initialized before (line 2). For example, the initial value of uninitialized parameter x is X, where X is the initial symbolic value of the parameter x.

Next, all statements of this test path will be analyzed in sequential order from the first statement (line 3). Each statement, denoted by *stm* in the test path, is rewritten into a simpler statement which makes symbolic execution become easier (line 4). This step is called *simplification process*, which is based on the inspiration of CIL [1]. In this process, the content of *stm* is modified based on memory model M and symbolic map S. The variables, which are used in *stm*, are replaced with its corresponding values or its corresponding addresses. Algorithm 5 will delve into the simplification process.

After the statement *stm* is rewritten, based on the type of statement *stm*, an appropriate analysis will be performed. There are four basic cases including *assignment*, *declaration*, *condition*, and *scope* as follows:

— *Case* 1: *Condition* (lines 5–11): Note that the condition is normalized in step 4 in which the used variables are replaced with its corresponding values or addresses. If the condition is *false*, it means that *PC* has no solution. In this case, the symbolic execution is terminated. Otherwise, the normalized condition will be added to path constraints *PC*.

— *Case* 2: *Declaration* (lines 12–21). For simplicity, our assumption is that the declaration does not has any assignment. In this case, the declared variable v will be added to memory model M and symbolic map S. The default value of a numerical variable, a pointer, and a string are 0, *NULL*, and "", respectively. The default value of a pointer is *NULL* because the pointer is not initialized in the declaration. The scope of the declared variable v is stored.

— *Case* 3: *Assignment* (lines 22–37). There are two main kinds of assignment depending on the type of the assigned variable (i.e. pointer, primitive variable). The value of the assigned variable e_1 in memory model M and symbolic map S will be updated. In the first case, the assigned variable e_1 is a pointer, if there exists an allocation of a pointer e_1 with size n (line 24), the assigned variable e_1 is updated in M at the given address (line 25). In addition, the initial value of pointer elements in e_1 are initialized to 0 (line 26). The constraint about the size of the pointer e_1 must be added to *PC* (i.e. $n >= 0$) (line 27). In the second case, the variable e_1 is assigned to *NULL* (line 28). Memory model M will update the new address of e_1 (line 29). All of the pointer elements of e_1 are removed from S (line 30). In the last case, the pointer e_1 is assigned to another pointer, namely e_2 from a specified location of the block where e_2 points to (lines 31–33). The starting location on this block is denoted by k ($k \in \mathbb{Z}$).

— *Case* 4: *Scope* (lines 28–35). The algorithm uses a variable named *scope* to store the scope of variables. Here, *scope* is used to store the level of locality of variables. The level of locality of a variable is used to locate where a variable is created in

the program. The level of locality of a parameter is always set to 0. For a local variable, its level of locality is always equal to 1. The value of *scope* will be decreased or increased when the statement *stm* is a closing scope or an opening scope, respectively. When the statement *stm* is a closing scope of a control block {...}, it means that all of the variables created in this block should be removed because these variables will not be used in the remaining part of the test path. In this case, memory model M and symbolic map S will remove these local variables. In order to detect which variables will be removed, Algorithm 4 will check the level of locality of each variable and will remove the redundant variables having the highest value of scope from M and S.

5.1.2. *Rewrite a statement*

Rewriting a statement, or simplification process, aims to transform a statement into a simpler form by replacing the used variables in this statement with its symbolic values or addresses. Algorithm 5 presents the rewritten procedure *rewrite*() mentioned in Algorithm 4. The input of Algorithm 5 includes the statement needed to be rewritten *stm*, memory model M, and symbolic map S. The output is the rewritten statement, denoted by *rewritten_stm*. The main idea of the simplification process is to modify the AST of the statement *stm* rather than performing on the original *stm* to avoid the problem when manipulating on the string. When the rewritten process is done, the AST will be converted into a corresponding expression. This expression is the shorten statement of *stm*.

Procedure *replace_variables*(e, M, S) replaces the variables used in the expression e with its symbolic values stored in symbolic map S or its addresses stored in memory model M. The purpose of the procedure $AST(e)$ is to get the AST of the expression e. Procedure *to_expression*(ast) returns the expression of the AST tree *ast*. Procedure *evaluate_boolean*(e_1, e_2, op) returns the boolean value of a binary expression $e_1\langle op\rangle e_2$ if it is comparable (i.e. *true*, *false*), where $op \in \{>=, >, <=, >, ==, !=\}$. Based on the type of the statement *stm*, the rewritten process is taken as follows:

— *Case* 1: *Assignment* (lines 1–10). The right-hand side of the statement *stm*, denoted by e_2, will be rewritten. Initially, the right-hand side is converted to its corresponding AST by using the procedure $AST()$. Next, the replacement of variables is performed on this AST (line 2). Finally, the newly modified AST of the right-hand side will be converted into an expression (line 3). In the case which the left-hand side of the statement *stm*, denoted by e_1, is an array item, the indexes of e_1 are rewritten similarly to the right-hand side (lines 4–8). The final rewritten statement *rewritten_stm* is created by merging the two rewritten expressions e_1' and e_2' (line 10).

— *Case* 2: *Condition type* 1 (lines 11–20). This is the comparison between the two expressions not related to *NULL*. Similar to the assignment case, the ASTs of the two sides of *stm* are constructed. After that, from the state of the variables stored in memory model M and symbolic map S, these two ASTs will be modified by

replacing the variables used in these trees with its corresponding values/addresses (lines 12 and 13). This step is done by using the procedure *replace_variables*(). Next, each of these two modified ASTs is exported to an expression, denoted by e'_1 and e'_2 (lines 14 and 15). The rewritten condition, denoted by *rewritten_stm*, evaluates its boolean value (lines 16–19). If the written condition *rewritten_stm* cannot be evaluated whether *true* or *false*, it means that this condition will still has variables inside and will added to the path constraints *PC* later.

— *Case* 3: *Condition type* 2 (lines 21–22). This is the comparison between the two sides in which one side is *NULL* and the other is a pointer. The pointer e_1 used in this statement is replaced with *NULL* if this pointer is not allocated or *not* − *NULL* if this pointer is allocated.

Algorithm 5. Simplification process of a statement: Rewrite()

Input: *stm*: a statement on a test path, M: memory model, S: symbolic map
Output: *rewritten_stm*: the shorten statement of *stm*

1: **if** $stm \equiv \mu(e_1, e_2, =)$ **then**
2: $AST_2 = replace\_variables(AST(e_2), M, S)$
3: $e'_2 = to\_expression(AST_2)$
4: **if** $e_1 \equiv$ an array item **then**
5: $index_1 =$ indexes of e_1
6: $var_1 =$ name of array variable in e_1
7: $AST_1 = replace\_variables(AST(index_1), M, S)$
8: $e'_1 = merge(var_1, to\_expression(AST_1)$
9: **end if**
10: $rewritten\_stm = \mu(e'_1, e'_2, =)$
11: **else if** $stm \equiv \mu(e_1, e_2 | op \in \{>=, >, <=, <, ==, !=\}, e_2 != NULL)$ **then**
12: $AST_1 = replace\_variables(AST(e_1), M, S)$
13: $AST_2 = replace\_variables(AST(e_2), M, S)$
14: $e'_1 = to\_expression(AST_1)$
15: $e'_2 = to\_expression(AST_2)$
16: **if** $e'_1 \equiv constant \&\& e'_2 \equiv constant$ **then**
17: $rewritten\_stm = evaluate\_boolean(e'_1, e'_2, op)$
18: **else**
19: $rewritten\_stm = \mu(e'_1, e'_2, op)$
20: **end if**
21: **else if** $stm \equiv \mu(e_1, NULL, op | op \in \{==, !=\})$ **then**
22: $rewritten\_stm = evaluate\_boolean(M(e_1), NULL, op)$
23: **else if** $stm \equiv$ a boolean variable v **then**
24: $rewritten\_stm = evaluate\_boolean(S(v), true, ==)$
25: **end if**
26: **return** *rewritten_stm*

— *Case* 4: *Condition type* 3 (lines 23–24). The condition is made of a boolean variable, denoted by v. The corresponding boolean value of v will be obtained from symbolic map S. The boolean value of the condition is evaluated in the procedure *evaluate_boolean*().

For example, given a statement $stm_0 \equiv a[1 + a[x]] > a[3]$, or $\mu(a[1 + a[x]], a[3], >)$ in which the state of memory model M and symbolic map S before parsing this statement is as follows:

$$S = \{(x, 2), (a[2], 0), (a[1], 4), (a[3], 10)\},$$
$$M = \{(addr(a), a), (addr(x), x)\}.$$

Here, $e_1 \equiv a[1 + a[x]]$, $e_2 \equiv a[3]$. This statement has the problem *implicit usage of variables* which is discussed later. In order to obtain the rewritten statement, initially, the ASTs of both the left-hand side e_1 and the right-hand side e_2 are built, namely AST_1 and AST_2, respectively. The partial AST of the expression stm_0 generated from CDT plugin is listed in Listing 2. Each node in this AST tree includes two parts: the content of the AST node and its type (e.g. *CPPASTArraySubscriptExpression*, *CPPASTBinaryExpression*, etc.).

Listing 2. The partial AST of stm_0 on rewrite() procedure (before replacement).

```
a[1 + a[x]] > a[3]: CPPASTBinaryExpression
    a[1 + a[x]]: CPPASTArraySubscriptExpression (first AST)
        a: CPPASTIdExpression
        1 + a[x]: CPPASTBinaryExpression
            1: CPPASTLiteralExpression
            a[x]: CPPASTArraySubscriptExpression
                a: CPPASTIdExpression
    a[3]: CPPASTArraySubscriptExpression          (second AST)
        a: CPPASTIdExpression
        3: CPPASTLiteralExpression
```

The second step in this procedure is to replace the used variables in the two ASTs, which are stored in memory model M and symbolic map S, with its corresponding symbolic values and its addresses, respectively. Here, we see that the variable x should be used in the first replacement step rather than array item of a. After the first replacement of x on the two ASTs, the corresponding condition will become

$$stm_1 \equiv a[1 + a[2]] > a[3].$$

In the two modified ASTs, both variables $a[2]$ and $a[3]$ can be replaced in the next step. After the replacement of $a[2]$ and $a[3]$ on the two ASTs, the corresponding condition is rewritten as follows:

$$stm_2 \equiv a[1 + 0] > 10 \quad \text{or} \quad a[1] > 10.$$

Next, after the replacement of $a[1]$ on the AST of the left-hand side, the corresponding AST tree of the expression is as follows:

Listing 3. The AST of example on rewrite() procedure (final replacement).

```
4>10: CPPASTBinaryExpression
    4: CPPASTLiteralExpression
   10: CPPASTLiteralExpression
```

Here is the expression corresponding to the above AST:

$$stm_3 \equiv 4 > 10.$$

The condition stm_3 is evaluated whether *true* or *false* by using the procedure $evaluate\_boolean(4, 10, >)$. The boolean value of the rewritten statement is 0 (i.e. *false*). This condition does not need to make simplification anymore.

5.1.3. *Discussion*

There are two main problems in symbolic execution including *name resolution* and *the implicit usage of variables*. Our analyzer solves these two main problems partially as follows:

— *Name resolution*: Our analyzer needs to detect the semantics of tokens in *stm* and its related information including declaration, definition, and references. For example, given the statement $int\ x = test(classA.getX())$, this statement has four primary tokens including *int*, x, *test*, and $classA.getX$. The symbolic execution engine needs to detect which ones in four tokens are variables, attributes, functions, etc. as well as its definitions/references. Our analyzer solves this problem by converting the testing function *fn* into the corresponding AST. This AST containing information about name resolution can be collected. The AST generation from the testing function *fn* could be performed by using GCC, Clang, or CDT plugin.[c] After that, the AST of the testing function *fn* will be traversed to collect necessary information related to name resolution.

— *Implicit usage of variables*: The used variables in *stm* are used implicitly that make the simplification process take more cost compared to the explicit usage. For example, considering the statement $int\ x = a[a[y]]$, we assume that $a[1] = 2$, $a[2] = 0$, and $y = 1$. In order to specify the value of variable x, our analyzer performs a simplification process as described in line 4 of Algorithm 4. In this process, the right-hand side is repeatably rewritten under a number of iterations. In an iteration, the name of variables is replaced with its values stored in symbolic map S or its address stored in memory model M. In this example, three iterations are good enough to make *stm* become simplest (i.e. no need for simplification any more). After the first iteration, *stm* becomes $int\ x = a[a[1]]$;

[c] https://www.eclipse.org/cdt/.

after the second one, it is $int\ x = a[2]$; and it becomes $int\ x = 0$ after the third iteration.

5.2. *SMT-Lib generation*

The main objective here is to apply capacity of various SMT-Solvers such as Z3, SMT-Interpol, etc. to solve path constraints automatically. Path constraints are usually represented in the form of logic expressions consisting of operands (i.e. contain array/pointer access or not), arithmetic operators (i.e. "+", "-", "*", "/", "%"), logical operators, comparisons, and negation ("!"). However, most of the modern SMT-Solvers have accepted the inputs satisfying SMT-Lib format. In other words, the format of path constraints, which is called logic expression, is incompatible with the input format of SMT-Solvers. Therefore, this paper presents the process of SMT-Lib generation from logic expressions in Fig. 3. First, each constraint, which is called a logic expression, in the given path constraints is transformed into a corresponding postfix expression. Next, the postfix expression is analyzed to obtain a corresponding expression tree. Finally, the expression tree is traversed to generate a corresponding SMT-LIB expression.

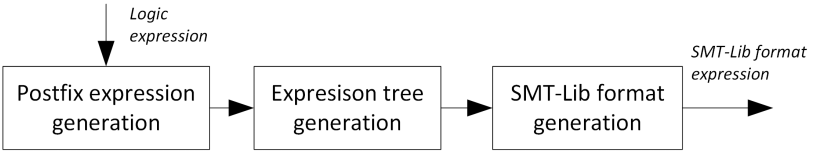

Fig. 3. The process of converting a logic expression into an SMT-Lib expression.

An example of the transformation is depicted in Fig. 4. The input is a logic expression $!(c > 0 || y < 1)$. Employing the transformation, this expression is converted into this postfix expression $x0 > y1 < ||!$. After that, this postfix expression is analyzed to construct a tree. In this tree, the first node (i.e. "!") represents the negation meaning. The leaves of this tree are operands. The parent of each leaf is corresponding to an operator. Next, an SMT-Lib expression is created by traversing

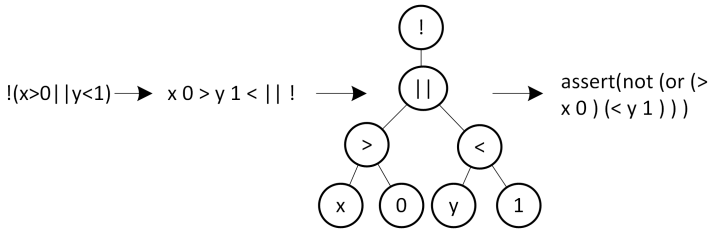

Fig. 4. An example of SMT-Lib expression generation from path constraints.

this tree. After that, this SMT-Lib expression is passed into an SMT-Solver to seek an appropriate solution.

5.3. *C++ general test driver generation*

Figure 5 presents the overview of improvement test data execution. The input includes an instrumented function and the location of an external file which stores previously generated test data. This function may be defined in a class/namespace or not. The process for creating a general test driver is described as follows. Initially, a basic test driver is created automatically which contains several operations such as reading content from file, etc. In later steps, this basic test driver will be added extra code for analyzing the structure of test data stored in an external file for the purpose of test data initialization.

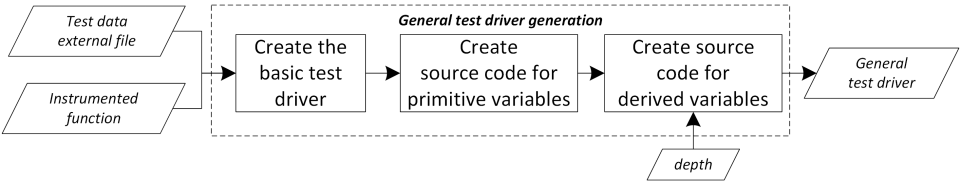

Fig. 5. General test driver generation on C++.

For primitive variables, the corresponding source code for loading these variables is created. In terms of derived types, the source code for each of these variables is produced based on the configuration parameters presented in Listing 4. In this list, the role of the parameter *DEPTH_LINKED_LIST* aims at specifying the maximum depth of a linked list to avoid the infinite construction problem. Parameter *MAX_RECURSIVE* stipulates the maximum of recursive iteration. Parameters *DEFAULT_VALUE_FOR_NUMBER* and *DEFAULT_VALUE_FOR_CHARACTER* present the default value of numbers and that of characters during initialization, respectively. The output of the proposed technique is a general test driver which can treat all of the values of primitive/derived types.

Listing 4. Default configuration in the improvement test driver generation.

```
static int DEPTH_LINKED_LIST = 4; //by default
static int MAX_RECURSIVE = 5; //by default
static int DEFAULT_VALUE_FOR_NUMBER = 0; //by default
static int DEFAULT_VALUE_FOR_CHARACTER = 58; //by default
```

The unique external file storing a test data is structured in the format of continuous lines in which each line represents a portion of test data. A portion of a test

data stored in the external file is defined as a pair TD as follows:

$$TD = (key, val),$$

where

- key is a parameter of the testing function fn, an element of a parameter as array/pointer, or a file of a parameter as struct/class, and
- val is the corresponding value of key. The type of val is a number, a character, or a string. If the type of a parameter is a pointer, its value may be $NULL$ or not. In the case if a parameter is a pointer, its size is presented in keyword $sizeof$.

Listing 5 depicts an example of *Student* structure initialization from an external file dynamically. Specifically, for each attribute in the definition of class *Student*, the corresponding source code for reading its value from file is created. Look at the first attribute *age* in *Student*, the template function *findValueByName* takes responsibility for reading the value of *age* (line 2). Concerning the pointer *s.name* (*char**), its size is initialized through its name by the template function *initializePointerByName* (line 3). In the external test data file, the variable *s.name* is allocated 10 bytes by default if its size is not set. The remaining code sample aims at constructing the value of attributes *id* and *address*.

Listing 5. An example of student initialization in the general test driver.

```
Student createStructureStudentByName(string nameStructure) {
1.  Student s;
2.  s.age = findValueByName<int>(nameStructure + ''.age'');
3.  s.name = initializePointerByName<string>(nameStructure + ''.name'');
5.  s.id = findValueByName<int>(nameStructure + ''.id'');
6.  s.address =
          initializePointerByName<string>(nameStructure + ''.address'');
7.  return instance;
}
```

5.4. *Control flow graph generation*

Control flow graph plays an important role in evaluating code coverage with a series of test data. In addition, this graph is also used to detect next partial test paths so as to generate a new test data which could increase code coverage.

Given a function under test, a decision of the given function is the condition of the control statement. A decision is made up of atomic logic expressions. An atomic logic expression is a logic expression which cannot be divided into logic expressions. For example, considering $if(a > b \&\& (a > c || b > c)) \{...\}$, the corresponding decision is $(a > b \&\& (a > c || b > c))$. The atomic logic expression set includes three logic expressions $a > b$, $a > c$, and $b > c$.

There are three types of code coverage criterion used widely in test data generation including statement coverage, branch coverage, and MC/DC coverage. A series of test data satisfying statement coverage criterion must visit all statements in the

given function. In the case where a series of test data traverses all branches of the decisions in the testing function (i.e. true branch, false branch), this test data set satisfies branch coverage criterion. The MC/DC coverage criterion is similar to branch coverage criterion. Generating a series of test data satisfying MC/DC coverage criterion requires that this test data set must visit all branches of atomic logic expressions. It can be seen that if the series of test data satisfies MC/DC coverage criterion, this test data set will satisfy branch coverage criterion.

Algorithm 6. Control Flow Graph Generation

Input: *currentBlockAST*: a block, *CFG*: a control flow graph, *cov*: coverage
Output: *CFG*: control flow graph

```
1: Initialize partial_AST = construct the AST of currentBlockAST
2: Set blocks = break partial_AST into blocks
3: Graph link_blocks = create links between elements of blocks
4: Replace currentBlockAST with link_blocks in CFG
5: for block: blocks do
6:    if (block is a control block ||(block is a condition && cov = MC/DC) then
7:       ControlFlowGraphGeneration(block, CFG, cov)
8:    end if
9: end for
10: return CFG
```

Algorithm 6 depicts the process of control flow graph construction from the function named *fn*. This algorithm takes three inputs including a block (*currentBlockAST*), a control flow graph (*CFG*), and a code coverage criterion (*cov*). *currentBlockAST* is an AST of an element in the testing function *fn*. The main idea of the algorithm is to divide blocks into smaller parts until its CFG satisfies code coverage criterion denoted by *cov*. In essence, the process of CFG generation is performed recursively. The analysis step starts with the AST of the testing function *fn*, and then tries to break this AST into smaller portions. After each of divisions, depending on the type of portions and *cov*, this portion will be continued splitting or not.

An example of control flow graph which satisfies statement/branch coverage is presented in Fig. 6. The beginning node and termination point are portrayed with a white circle and a black circle, respectively. Generally, given a function, the control flow graph corresponding to MC/DC coverage criterion might be more complex than that of statement/branch coverage criterion. The main reason is that the former requires the analysis of conditions, while the latter does not need it.

Figure 6 illustrates the corresponding control flow graph of function *Utils*::*Fibonaxi*. A partial AST of function *Utils*::*Fibonaxi* is described in Listing 6. Each AST element belongs to a specific type such as *CPPASTIfStatement* (*If* blocks), *CPPASTBinaryExpression* (binary expressions), *CPPASTFunctionCallExpression*

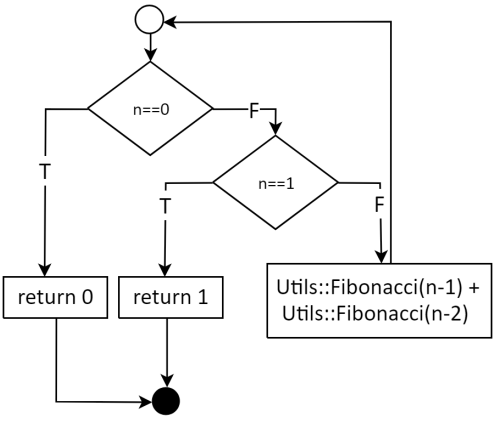

Fig. 6. A control flow graph of function *Utils*::*Fibonaxi*.

Listing 6. A partial AST of function *Fibonaxi*.

```
int y = Algorithm::Utils::Fibonacci(n-2);: CPPASTDeclarationStatement
    int y = Algorithm::Utils::Fibonacci(n-2);: CPPASTSimpleDeclaration
        int: CPPASTSimpleDeclSpecifier
        y = Algorithm::Utils::Fibonacci(n-2): CPPASTDeclarator
            y: CPPASTName
            Algorithm::Utils::Fibonacci(n-2): CPPASTEqualsInitializer
                Algorithm::Utils::Fibonacci(n-2): CPPASTFunctionCallExpression
                    Algorithm::Utils::Fibonacci: CPPASTIdExpression
                        Algorithm::Utils::Fibonacci: CPPASTQualifiedName
                    n-2: CPPASTBinaryExpression
```

(function calls), or *CPPASTName* (corresponding to the name of variable, function, etc.).

5.5. *Code coverage computation*

During test data execution, the content of a statement will be appended at the end of an external file when it is executed on an instrumented function fn' of a function fn. Code coverage is then computed to detect whether the test data are useless or not. Algorithm 7 presents code coverage computation in detail. Starting with the CFG of the testing function fn satisfying branch coverage criterion and a test path, Algorithm 7 will traverse along this CFG to update the visited state of statements/ branches. Initially, the set *recursivePoints*, used to store recursive sites in the given test path, is initialized to empty (line 1). After that, *currentNode* is set to the beginning node of the CFG (line 2). The process will analyze all statements in *testpath* basically as follows.

Consider the first case where *currentNode* is *EndNode*, the ending node of CFG, there occurred two cases. Specifically, the termination of the current call (namely C_1)

Algorithm 7. Code Coverage Computation

Input: tp : a test path of a function fn, CFG: the CFG of a function fn
Output: CFG: the updated CFG

```
1:  Stack<CfgNode>recursivePoints = {}
2:  CfgNode currentNode = CFG.beginNode
3:  for Statement stm: tp do
4:    if currentNode is EndNode then
5:      if recursivePoints.size >= 1 then
6:        currentNode = recursivePoints.pop().nextNode
7:      end if
8:    else if currentNode is a recursive call then
9:      recursivePoints.push(currentNode)
10:     currentNode = CFG.beginNode
11:   else if currentNode is a condition then
12:     if nextStm is currentNode.trueNode then
13:       setVisitedBranch(currentNode, currentNode.trueNode)
14:       currentNode = currentNode.trueNode
15:     else
16:       setVisitedBranch(currentNode, currentNode.falseNode)
17:       currentNode = currentNode.falseNode
18:     end if
19:   else
20:     setVisitedStatement(currentNode)
21:     currentNode = currentNode.nextNode
22:   end if
23: end for
```

might be the start of another call (namely C_2) because of recursive call; or the program terminates completely. When the first case happens, *currentNode* will point to the next executed statement where C_1 happens (lines 4–7).

In the second case statement that *stm* contains a recursive call, *currentNode* is pointed to the beginning of *CFG*. *currentNode* is set to the beginning of CFG (lines 8–10). The main reason is that, when a recursive call is performed, the testing function *fn* will be executed in another stack.

In other cases that *stm* is a condition or a simple statement (e.g. assignment, declaration, **return**), the state of nodes corresponding to visited statements/branches are updated (lines 11–21).

For example, considering the test path "{=>n == 0=>n == 1=> return (Fibonacci($n-1$) + Fibonacci(n−2)); (*)=> {=> **n == 0=>n == 1=> return 1;**=>{=> n == 0=>return 0;" generated from executing function *Fibonacci* in Listing 1. This function contains two recursive calls *Fibonacci*$(n-1)$ and *Fibonacci*$(n-2)$. The bold statements are corresponding to the executed statements

of the first call while the italic ones belong to the results of the second call. At the statement denoted by (*), because there are two recursive calls, two recursive points are created in *recursivePoints* that $recursivePoints = \{(*).nextNode, (*).nextNode\}$. On this example, (*).*nextNode* is equivalent to the end node of CFG. When the first recursive call is performed, the testing function is executed in another stack. Therefore, *currentNode* will be pointed to the starting node of *CFG*. After executing bold statements, the current call terminates, then *currentNode* continues moving to the beginning point of the second call $Fibonacci(n-2)$.

6. Experiments

6.1. *The implemented tool*

Our proposed SDART has been implemented in a tool named ICFT4Cpp[d] using Java to demonstrate its effectiveness. Our objective is to demonstrate the efficacy of SDART in comparison with DART with the same core implementation (e.g. symbolic execution engine). There are several differences between the implementation of DART and SDART. However, these differences do not affect the accuracy of the experiment. Specifically, rather than applying DSE, we try to record executed statements/branches during test data execution and then stored it in an external file. When the testing function raises an exception or successes, the test path in this file is loaded to perform symbolic execution.

Another difference is that ICFT4Cpp does not evaluate code coverage during test data execution. Instead, only when a test path is recorded successfully, code coverage is computed immediately. This code coverage computation differs from that of DART, which evaluates during test data execution. In DART, the process of evaluating the boolean value of condition depends completely upon the function *evaluate_concrete* which might be incorrect due to bad implementation. In contrast, we simply use execution results to check whether the value of a condition *true* or *false*. Therefore, there is no mistake in this step.

Figure 7 presents the architecture of ICFTCpp. In brief, the architecture of ICFT4Cpp is similar to CFT4Cpp, which is proposed in [23]. There are two main layers including presentation layer and logic layer. The logic layer takes responsibility for interacting with Z3 SMT-Solver [4] and MingW32 compiler. The implemented tool uses plug-in Eclipse CDT to assist in syntax analysis [20] and mcpp library for removing preprocessor.[e] Eclipse CDT supports name resolution directly in its AST so it is easy to resolve the properties of a variable, e.g. its definition or its references.

6.2. *Experimental results*

The experiment is performed on a Windows machine with an Intel(R) Core(TM) i5-4200U CPU @1.6 GHz–2.3 GHz using Mingw32 with 12GB RAM. The testing C+

[d] https://github.com/ducanhnguyen/cft4cpp-core.
[e] http://mcpp.sourceforge.net/.

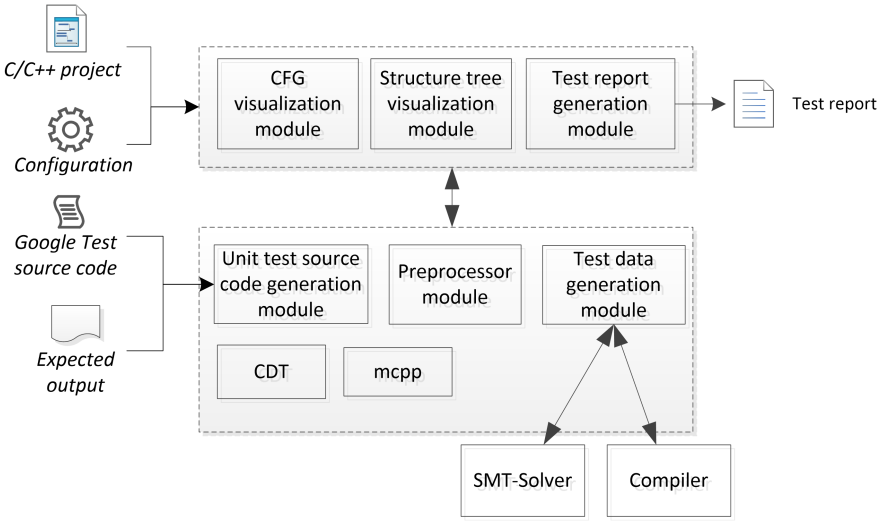

Fig. 7. The architecture of ICFT4Cpp.

+ project is created as follows. First, we collected functions from various websites, i.e. *programmingsimplified.com*, *geeksforgeeks.org*, and *pathcrawler-online.com*. A function is selected if it satisfies two criteria: (1) it must be compatible with our implementation and (2) it should be related to algorithm. Here, a function is compatible if it is fully supported by our symbolic execution engine. Choosing the functions related to algorithm ensures that the select functions have high level of complexity. As a result, 51 functions were found in total and the number of lines of code adds up to 1000. Most of the selected functions are related to algorithms such as *BubbleSort*, *Merge*, etc. In the second step, these functions were put into a project created by IDE Dev-Cpp. We choose this IDE because Dev-Cpp[f] is a popular IDE for C/C++ and used widely in universities.

In this experiment, the comparison between DART and SDART is carried out under the same configurations as follows:

- **DART.** The depth of DART is initialized to 3. At each depth, at most 10 iterations will be performed. In total, there are at most 30 iterations during test data generation for each function. The bound of integer variables is [0...30]. The bound of character is set in the visible range. The path selection strategy of DART used in this experiment is BFS.
- **SDART.** The maximum iteration for each function in SDART is equivalent to that of DART (i.e. 30 iterations). The configuration of variables in SDART is similar to DART. The threshold to switch to the static test data generation mode is four. It means that if there exist four continuously random test data which do

[f]https://www.bloodshed.net/devcpp.html.

not increase code coverage, the static test data generation mode will be used instead.

With the given configuration, it could be seen that in the worst case, both DART and SDART will take at most 51 $functions * 30$ $iterations = 1530$ $iterations$. Also, in the best case, both strategies will perform at least 51 iterations in total. It also means that the first random test data will traverse all branches.

In Fig. 8, it can be seen that SDART tends to achieve the higher number of visited branches compared to DART. Specifically, although SDART does not show its effectiveness in about 120 beginning iterations, SDART gradually surpasses DART in the remaining ones. While DART takes around 410 iterations in total, SDART only executes these testing functions with approximately 370 iterations. The main reason is when SDART detects the low possibility of increasing code coverage, it switches to the static test data generation. With this strategy, SDART expects that more branches will be found earlier than keeping the current path selection strategy.

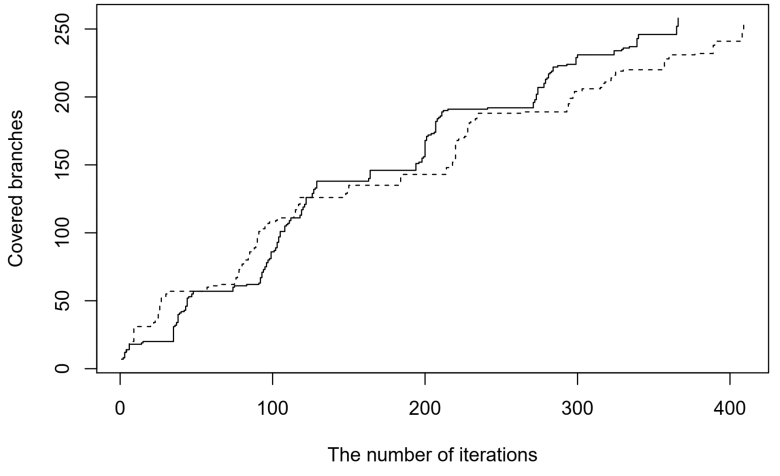

Fig. 8. The comparison between DART (BFS) and SDART (threshold = 4) in terms of visited branches. DART-BFS and SDART (threshold = 4) are represented in a dashed line and a solid line, respectively.

More details about the comparison between DART and SDART are presented in Table 3. First, the total number of solver calls in DART is considerably greater than that of SDART, with 1279 solver calls and 413 solver calls, respectively. The main reason is that DART tends to generate the next test data by negating the conditions which are seem to be useless in terms of increasing code coverage. Second, SDART generates a larger number of meaningful test data with a smaller number of iterations in comparison with DART. Here, test data are considered as a meaningful test data when the test data increase code coverage. Specifically, the meaningful test data

Table 3. Information about test data generation in DART (BFS) and SDART.

	SDART	DART (BFS)
Number of solver calls	413	1279
Number of symbolic statements	12,922	16,167
Number of meaningful test data	101	102
Number of iterations	366	409
Branch coverage	**93.48%**	92.03%
Meaningful test data rate	**27.6%**	24.94%

generated in SDART accounts for 101/366 * 100% = 27.6%, where 101 and 366 are the numbers of meaningful test data and the number of iterations, respectively. This percentage in DART is just 102/409 * 100% = 24.94%. Whenever SDART detects, it is less likely to increase code coverage under a number of iterations, SDART will switch to the static test data generation. SDART expects that this static test data generation will generate a series of newly meaningful test data.

7. Conclusion

This paper presented two improvements for tackling the problems related to the number of test data and the computational cost of test data compilation on C++ projects. The first improvement named SDART aims to increase code coverage as fast as possible by combining the BFS strategy of DART with the static test data generation approach. Specifically, whenever it is less likely to generate a new test data increasing coverage, the static test data generation will be chosen instead. A list of partial test paths covering unvisited branches will be selected by analyzing CFG. The cost for analyzing these partial test paths in symbolic execution engine is reduced because these paths are made up of a smaller number of statements. Second, concerning the computational cost of test data compilation, the proposal tries to generalize a C++ test driver to deal with various data types to reduce the computational cost of test data compilation on C++ projects. In order to do that, the proposed method extends the DART to deal with C++ projects. All of the test data are stored in an external file and a C++ general test driver takes responsibility for reading these values to initialize the parameters of a testing function dynamically.

The experiment has shown that the proposed SDART only takes a smaller number of iterations with the smaller number of iterations while gaining higher branch coverage in comparison with the BFS strategy of DART. Currently, the proposed method has been implemented in the tool named ICFT4Cpp. This tool is currently used in Toshiba Software Development Vietnam company and received many positive feedbacks. Based on these suggestions, the proposal will be improved to deal with more C++ features used widely in industrial projects. First, generating test data for templates and polymorphism is still considered as a big challenge. The main reason is that it is difficult to detect exactly the functions which are called

during execution. Second, the proposal will extend to generate test data for the functions containing exceptions. Specifically, the research will extend to produce a series of test data which cause runtime errors. Third, the C++ general test driver will be improved to support more various types in C++ such as vector, list, etc. Finally, the symbolic execution needs to be improved to analyze the statements using overloading mechanism. Specifically, both *memory model* and *symbolic map* should be enhanced to deal with overloading, e.g. the subtraction of two class instances.

Acknowledgments

This work is supported by the research project No. QG.16.31 granted by Vietnam National University, Hanoi (VNU).

References

1. G. C. Necula, S. McPeak, S. P. Rahul and W. Weimer, CIL: Intermediate language and tools for analysis and transformation of C programs, in *Proc. 11th Int. Conf. Compiler Construction*, 2002, pp. 213–228.
2. J. C. King, Symbolic execution and program testing, *Commun. ACM* **19** (1976) 385–394.
3. C. Cadar and K. Sen, Symbolic execution for software testing: Three decades later, *Commun. ACM* **56** (2013) 82–90.
4. L. De Moura and N. Bjrner, Z3: An efficient SMT solver, in *Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 337–340.
5. G. Li, I. Ghosh and S. P. Rajan, KLOVER: A symbolic execution and automatic test generation tool for C++ programs, in *Proc. 23rd Int. Conf. Computer Aided Verification*, 2011, pp. 609–615.
6. C. Cadar, D. Dunbar and D. Engler, KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs, in *Proc. 8th USENIX Conf. Operating Systems Design and Implementation*, 2008, pp. 209–224.
7. D. Riehle, Composite design patterns, in *Proc. 12th ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, 1997, pp. 218–228.
8. Z. Wang, X. Yu, T. Sun, G. Pu, Z. Ding and J. Hu, Test data generation for derived types in C program, in *Third IEEE Int. Symp. Theoretical Aspects of Software Engineering*, 2009, pp. 155–162.
9. N. Williams, B. Marre, P. Mouy and M. Roger, PathCrawler: Automatic generation of path tests by combining static and dynamic analysis, in *Proc. 5th European Conf. Dependable Computing*, 2005, pp. 281–292.
10. P. Godefroid, N. Klarlund and K. Sen, DART: Directed automated random testing, in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, 2005, pp. 213–223.
11. K. Sen, D. Marinov and G. Agha, CUTE: A concolic unit testing engine for C, in *Proc. 10th European Software Engineering Conf.* held jointly with *13th ACM SIGSOFT Int. Symp. Foundations of Software Engineering*, 2005, pp. 263–272.
12. T. Su *et al.*, Automated coverage-driven test data generation using dynamic symbolic execution, in *Eighth Int. Conf. Software Security and Reliability*, 2014, pp. 98–107.
13. J. Burnim and K. Sen, Heuristics for scalable dynamic test generation, in *Proc. 23rd IEEE/ACM Int. Conf. Automated Software Engineering*, 2008, pp. 443–446.

14. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill and D. R. Engler, EXE: Automatically generating inputs of death, in *Proc. 13th ACM Conf. Computer and Communications Security*, 2006, pp. 322–335.
15. D. M. Perry, A. Mattavelli, X. Zhang and C. Cadar, Accelerating array constraints in symbolic execution, in *Proc. 26th ACM SIGSOFT Int. Symp. Software Testing and Analysis*, 2017, pp. 68–78.
16. H. Palikareva and C. Cadar, Multi-solver support in symbolic execution, in *Proc. 25th Int. Conf. Computer Aided Verification*, 2013, pp. 53–68.
17. Z. Xu, T. Kremenek and J. Zhang, A memory model for static analysis of C programs, in *Proc. 4th Int. Conf. Leveraging Applications of Formal Methods, Verification, and Validation — Volume Part I*, 2010, pp. 535–548.
18. B. Elkarablieh, P. Godefroid and M. Y. Levin, Precise pointer reasoning for dynamic test generation, in *Proc. Eighteenth Int. Symp. Software Testing and Analysis*, 2009, pp. 129–140.
19. T. Sun, Z. Wang, G. Pu, X. Yu, Z. Qiu and B. Gu, Towards scalable compositional test generation, in *Ninth Int. Conf. Quality Software*, 2009, pp. 353–358.
20. D. Piatov, A. Janes, A. Sillitti and G. Succi, Using the eclipse C/C++ development tooling as a robust, fully functional, actively maintained, open source C++ parser, in *Open Source Systems: Long-Term Sustainability*, 2012, p. 399.
21. P. Godefroid, Compositional dynamic test generation, in *Proc. 34th Annual ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, 2007, pp. 47–54.
22. P. C. Jorgensen, *Software Testing: A Craftsman's Approach* (CRC Press, Boca Raton, 2014), pp. 6–9.
23. D.-A. Nguyen and P. N. Hung, A test data generation method for C/C++ projects, in *Proc. Eighth Int. Symp. Information and Communication Technology*, 2017, pp. 431–438.