2019 IEEE 37th VLSI Test Symposium (VTS)

A New Method for Software Test Data Generation Inspired by D-algorithm*

Jianwei Zhang, Sandeep K. Gupta, William G. J. Halfond
University of Southern California, Los Angeles, USA
Emails: {jianweiz, sandeep, halfond}@usc.edu

***Abstract*—Test generation for digital hardware is highly automated, scalable (in practice), and provides high test quality. In contrast, current software automatic test data generation approaches suffer from low test quality or high complexity. While *mutation*-oriented constraint-based test data generation for software was proposed to generate high quality test data for real program bugs, all existing approaches require symbolic analysis for the whole program, and hence are not scalable even for *unit testing*, i.e., testing the lowest-level software modules.**

We propose a new method inspired by hardware D-algorithm and divide and conquer for *software* test data generation. To reduce runtime complexity and improve scalability, we combine global structural analysis and a sequence of small reusable symbolic analyses of parts of the program, instead of symbolically executing each mutated version of the entire program. We also propose a multi-pass test generation system to further reduce runtime complexity and compact test data. We compare our tools with one of the best software test generation tools (EvoSuite[20], which won the SBST 2017 tool competition) and demonstrate that our approach generates higher quality unit tests in a scalable manner and provides a compact set of tests.

Keywords—test data generation for software; D-Algorithm; ATPG; mutation testing; software testing

I. Introduction and Background

A. Hardware Testing and Software Testing

In industry practice, hardware (HW) test generation is highly automated. Several test generation algorithms were developed and refined via five decades of research. The most common algorithms used for HW test generation, like D-algorithm [1] and PODEM [2], are fault model based. Research and practice show that a test set with high coverage of faults typically also provides high coverage of real hardware defects.

In software (SW) testing research, test generation automation is limited and less developed compared to HW, due to the complexities of data types and program structures. Test set quality is commonly measured by code coverage metrics [3], which capture the percentage of lines, branches, etc. covered by the test (also adapted for HW ATPG, e.g., [8]). However, their effectiveness as measure of test quality is doubtful [4], especially for real program errors deep within the program.

Demillo et al. [5] proposed the concept of a program *mutant*, i.e., a modified program that diverges from the original program, usually by a change at one statement. A mutant is marked as *killed* if we have a test for which the program's outcome (e.g., output values) for the mutant is different from that for the original program. *Mutation score*, i.e., the fraction of mutants killed from the complete set of mutants, captures test set quality. Research [6] shows a strong correlation between a high mutation score and a high coverage of real program bugs. (Mutation testing is also adapted for HW RTL design verification [19].)

HW fault-oriented testing and SW mutation testing are both based on abstract fault models and effective in identifying real defects/bugs. We find that they share many similarities and the major analogies between the two are listed in Table I.

----

* Research funded by National Science Foundation

TABLE I. Analogy Between HW Testing and SW Mutation Testing

	HW testing	**SW mutation testing**
Descriptive language	Netlist	Lines of code
Basic element	Gate	Statement(s)
Interconnections	Circuit line	CFG path, DU path
Defect/bug	Fault	Mutation
Defective artifact	Faulty circuit	Mutant
Common fault model	Single stuck-at fault	Single mutation

Further, in HW testing, shorter test sets are desired as they require smaller test application times and hence considerably decrease testing costs, since every fabricated chip needs to be tested. In contrast, SW testing is only performed once for a program, independent of how many copies are sold. Despite this, *a compact test set is extremely important in SW testing for a very different reason. Since a golden model of the software usually does not exist (test oracle problem [21]), each test's outcome must be manually checked for correctness*. Since such manual checking is extremely expensive, a compact set of tests that provides high quality can considerably reduce costs.

B. Test Generation for SW Mutant

Although the initial purpose of SW mutation testing was to find a more accurate way for evaluating the quality of a given test set, mutation-oriented test generation has been pursued to generate high quality tests. Demillo et al. [7] proposed the first method to generate a test for a targeted mutant by converting the whole program into a constraint system and using a constraint solver to generate a test. On the other hand, search-based test data generation was first suggested by Bottaci [10]. It models the test generation task as a search problem guided by a fitness function. In this category, EvoSuite [20] is one of the best state-of-the-art tools. Search-based methods suffer from the limitations of the corresponding heuristics.

In this paper, we focus on improving the constraint-based test generation approaches. All the existing methods in this category are not scalable even for *unit testing* (testing the lowest level SW modules), since they need to convert the entire program unit and conditions into constraints, which are then processed by a constraint solver. Further, the constraints are usually generated via symbolic execution [12], which is especially expensive. Also, the solver has high complexity since it faces long expressions. We address these shortcomings to make constraint-based test generation widely applicable for unit testing.

II. Main Ideas

We propose a new method for **unit testing** which makes mutation-oriented constraint-based test generation scalable yet accurate.

We use divide and conquer to reduce the complexity of constraint creation and solving. Specifically, we *combine global structural analysis and a sequence of small local (for parts of the program) symbolic analysis*, instead of performing symbolic analysis on the whole program. This reduces the load on constraint solver by deriving smaller constraint expressions. Also, structural analysis, which includes control flow analysis and data dependency analysis, is a low-complexity processes compared to symbolic analysis. The structural information it generates for the program under test connects the symbolic expressions for different parts of the program, thus significantly

```
0  void function (int a, int b, int c) {
1    int x = 0;
2    if (a == b) {
3      x = x + 1;
4    }
5    if (b == c) {
6      x = x + 1;
7    }
8    if (a == c) {
9      x = x + 1;
10   } else {
11     x = x + 2;
12   }
13   return x;
14 }
```

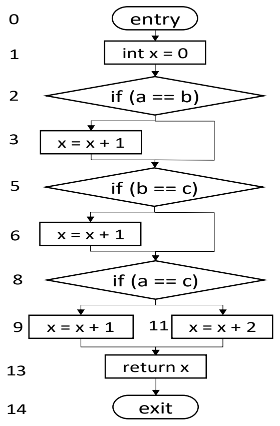

Fig. 1. An example program under test and its CFG

reduces runtime complexity compared to traditional methods, especially for large units and units with path explosion [22].

In addition, inspired by the concept of HW D-algorithm [1], we develop SW D-algorithm to restrict the generation process to directed local search. HW D-algorithm was the first complete test generation algorithm which established a paradigm for completely searching the space of all possible tests. Its test generation subtasks (TGSTs) include fault excitation, fault effect propagation, and justification. Similarly, our *new approach, SW D-algorithm, has three test generation subtasks: mutation effect excitation, mutation effect propagation, and justification*.

Moreover, we propose a multi-pass *test generation system*, which uses random test generation to cheaply kill mutants before applying deterministic SW D-algorithm. After we generate tests using our SW D-algorithm, we use *reverse order simulation* (adapted from HW test generation) to compact SW test data. The details are presented in Section IV.

III. Review of SW Analysis Methods

Before presenting the details of SW D-algorithm, some basic methods used in SW analysis are introduced for completeness.

A. *Structural Analysis of Programs*

Structural analysis is a *static method* for identifying control flow and data dependency for a program. (Static analysis methods have significantly lower run time complexities compared to dynamic methods, which require analysis/ simulations for large numbers of specific input values.) Our algorithm requires two types of static information of the program: the control flow graph (CFG) and all *define-use chains* (DU chains) in the program.

CFG is a directed graph, where each node represents a basic block, and each edge represents a path the control flow may follow. CFG is generated from program's code by traversing every statement. Fig. 1 shows an example Java code and its CFG.

During *static analysis* on CFG, following notions [13] are used: (1) *Dominator* – a node x *dominates* a node y if every path from the program's entry node to y passes through x; we call x as y's *immediate dominator* if x is y's closest *dominator*. (2) *Post-dominator* – a node x *post-dominates* a node y if every path from y to the program's exit node passes through x; *immediate post-dominator* is defined analogously. In the CFG shown in Fig. 1, node 2 is node 5's *immediate dominator*, node 8 is node 5's *immediate post-dominator*.

In SW, a variable v is said to have a *definition* at statement x if v is assigned a value at x; v is said to have a *use* at statement x if v determines either the value of another variable *defined* at statement x or determines the program flow in the case where the statement x is a *conditional* statement. In addition, we consider that the *definition* of variable v at statement x *reaches* statement y (*reaching definition*) if there is a path in the CFG from x to y that does *not* pass through any other *definitions* of v. We use the form of "*def/use (variable name, location where the variable is used/defined)*" to represent a variable's *definition* or *use* at the specified location. In Fig. 1, variable *a* defined at the entry node (*a* is a user input variable) is denoted as *def(a,0)*; variable *a* is used at node 2 and node 8, we represent them as *use(a,2)* and *use(a,8)*. Also, *def(a,0) reaches use(a,2)* and *use(a,8)*.

A *DU chain* consists of a *definition* of a variable and all its *uses*, which are reachable from that *definition*. In Fig. 1, *def(a,0)* and its *uses*, *use(a,2)* and *use(a,8)* form a DU chain of variable *a*.

B. *Symbolic Analysis of Programs*

In HW testing, BSF (*Boolean switching function*) is used to describe the relations between circuit's input and output logic variables. In SW testing, the code behavior can be described using a *symbolic expression* generated by *symbolic execution* [12]. To model a program using symbolic expression, we need to identify program's inputs and outputs. We use the notion of program's I/O streams to describe program's inputs from different sources and outputs to different destinations. These sources/destinations include disk files, user consoles, etc. The basic I/O streams, in JAVA for example [14], include byte streams, I/O from the command line, etc. In a program, we identify a set called *program's input definitions* (PID) and a set called *program's output definitions* (POD), which includes all *definitions* representing program's I/O streams. We identify PID and POD from program's specification. In Fig. 1, PID consists of *def(a,0)*, *def(b,0)* and *def(c,0)*; POD has the *definition* of the returned value at line 13, which is denoted as *def(retVal,13)*. We note that the value of *retVal* equals to the value of *x* at *use(x,13)*.

IV. Our Method: SW D-algorithm

SW D-algorithm is developed by building on the principles used in HW D-algorithm. But due to many differences between SW and HW, we must significantly extend several existing definitions and methods used in HW D-algorithm to adapt them to SW and develop new concepts and methods to capture special characteristics in SW. Due to the limited space, we cannot include all details and all possible cases; interested readers please refer to the following doctoral dissertation proposal [15].

A. *SW Representation for SW D-algorithm*

We need to convert SW program to a form that is suitable for SW D-algorithm. Here we describe all essential components we use to represent a SW program. We note that currently we use loop unwinding to convert cyclic programs to acyclic.

1) *Basic Block in SW – Minimal region of analysis (mROA)*

In a HW digital circuit ATPG, each gate or line is treated as a basic block since its behavior can be captured strictly in terms of values at its input(s) and output(s), with or without any fault inserted within the block. In a SW, this is true for every individual non-conditional statement. However, the behavior of a conditional statement cannot be captured solely in terms of values at the conditional statement's value-inputs/outputs alone, since the execution of the statement determines not only the values at its outputs, but also which branch is taken.

To tackle this complication, we developed the new notion of *minimal region of analysis* (mROA) which must contain the statement under study and a minimal number of additional statements. The goal is to confine any changes in the program flow (i.e., invocations) within the mROA. Here we use $mROA_s$ to represent a minimal region of analysis for a statement s. The behavior of this part of the program - with or without a mutation in the statement s - must be captured only in terms of values at *inputs and outputs* (defined ahead) of the $mROA_s$.

We identify $mROA_s$ for a statement s such that it has a unique entry node and a unique exit node on *CFG* and includes this statement. This allows us to capture $mROA_s$'s behavior by only monitoring the value changes at its outputs, without considering any program flow changes from/to the scope of the $mROA_s$. An $mROA$ is represented using the line numbers of its entry and exit nodes. For example, $mROA\ [x,y)$ contains all paths and nodes between node x and y on *CFG* except node y (parenthesis and square brackets are used, respectively, to denote whether the starting and ending line are included or excluded).

There are two cases for identifying $mROA_s$ of statement s: (1) s is a non-conditional statement; (2) s is a conditional statement. In the first case, $mROA_s$ is simply s itself. In the second case, $mROA_s$ contains all the nodes and paths between s and its immediate post-dominator (but exclusive of the immediate post-dominator) on *CFG*, since the mutation's presence may change program flow across different branches from that conditional statement. For example, in Fig. 1, since the statement 2 is a conditional statement, $mROA_2$ is [2,5).

The behavior of an mROA is represented using symbolic expressions on its local inputs and outputs. Similar to the PID and POD of the whole program, we define a set called *mROA's input definitions* (RID) as the set of all *definitions used* within the mROA but *defined* outside of the mROA, and a set called *mROA's output definitions* (ROD) as the set of all *definitions defined* within the mROA that *reach* outside of the mROA. Symbolic execution is used to generate mROA's symbolic expression, using variables in RID/ROD as input/output symbols.

By obtaining such a model of mROA, all statements within mROA can be viewed as a "single" statement from outside and their behavior captured completely by mROA's symbolic expression. For example, for the $mROA_2$ [2, 5) for line 2 in Fig. 1, symbolic execution will generate symbolic expression "$a = b\ \&\&\ x\_out = x + 1\ ||\ a\ != b\ \&\&\ x\_out = x$". (We use the "$\_out$" suffix is to indicate that it is mROA's output variable.)

2) Interconnections in SW

In HW circuits, gates are connected by circuit lines, which carry logic values and indicate flow of control via logic value transitions. Circuit lines are, hence, the sole interconnections in HW circuits, while in SW programs, there is no physical "line" between statements or between a DU pair in a DU chain. Instead, we use both control flow and data dependency information to fully capture the interconnections within SW.

The first type of interconnections in SW is represented as a sequence of *nodes* and *edges* in its CFG. The second type of interconnections is represented as *DU chains*. In Fig. 1, *def(x,1)* and *use(x,3)* are connected not only in terms of the path between node 1 and node 3 in *CFG* (path 1-2-3), but also in terms of the *DU path* between *def(x,1)* and *use(x,3)*.

3) Mutation effect

In HW D-algorithm, fault effect at a circuit line is denoted as D or $\overline{D}$ [1], which is based on multi-valued *composite value* system to describe different values in fault free and faulty versions of the circuit. Similarly, in SW D-algorithm, we define *mutation effect* as the presence of two different values for the same variable in the original program and its corresponding location in the mutant. The objective of test data generation is to excite the *mutation effect*, propagate the *mutation effect* to one or more program output, and ensure that all values are *justified.*

The data type in SW is more complex compared to Boolean values used in digital circuits, thus we cannot use a composite value system to represent values of variables in different versions of code. For example, if an integer variable has *mutation effect*, we cannot use a simple composite value system to represent all possible value combinations since original and faulty values can both take a very large number of possibilities within the value range of integer. Instead, we can use a combined representation: "*value of the variable in the original program / value of the variable in the mutant*". And *mutation effect* exists if the two values are different. In Fig.1, consider a mutant that changes the statement at line 3 from "$x = x + 1$" to "$x = x - 1$". We observe a *mutation effect* at this line if we assign x any integer, say 0. And the *mutation effect* can be denoted as *def(x,3) = 1/-1*.

To find the corresponding variables *used* or *defined* in the original program and the mutant, we create a mapping function φ for the correspondences between the statements in original program and the mutant. Given a line p in original program, we can find its corresponding line in the mutant as $\varphi(p)$.

4) Invocation status

In HW, every element in a digital circuit is always *invoked* due to its inherent parallelism, while in a SW program, a node (statement) or an edge in program's CFG is not automatically *invoked* until it is executed during runtime, because program flow branches after a conditional node, and only one branch is executed. The precondition of executing this branch is determined at runtime. We define *edge condition* (*EC*) as the precondition for a specific edge to be executed. If a statement is not invoked, any change induced by this statement, in control or data flow, must not be counted. This difference between HW and SW requires a completely new set of variables and algorithms.

We define *invocation status* (IVS) of a node/edge in a program's CFG to indicate whether this node/edge will be *invoked* during runtime. IVS is a new type of dynamic information proposed here, it is introduced to reflect the dynamic execution information of a node/edge during runtime. Thus, a *DU path* within a *DU chain* may be invalid when considering IVS values of edges/nodes along the *DU path*. Therefore, an *active DU path* is defined as a *DU path* such that all nodes/edges along this path are *invoked* or with unknown IVS, since we can set this IVS as invoked. An *active DU chain* is obtained by eliminating all *inactive DU paths* between *definition* and *use* pairs in the *DU chain*. Also, we say *def(x) actively reaches use(x)* if there exists at least one *active DU path* between *def(x)* and *use(x)*, and *def(x)* is called an *active reaching definition* of *use(x)*. In Fig. 1, static analysis shows *use(x,6)* has two reaching definitions: *def(x,1)* and *def(x,3)*. But if the *edge 2-3* is not *invoked* during runtime, only *def(x,1) actively reaches use(x,6)*.

B. Essential Procedures in SW D-algorithm

We now outline essential procedures in SW D-algorithm. Although these procedures are conceptually similar to their counterparts in HW D-algorithm, they are markedly extended and reinvented to include special characteristics of SW.

1) Implication

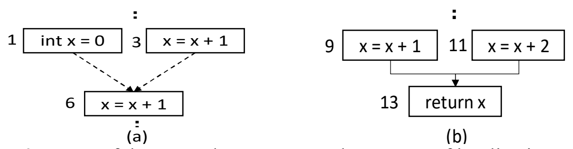

Fig. 2. Parts of the example program to show cases of implication

In general, implication is the process of determining some values as a consequence of the changes in some other values. Implication is performed to reduce the search space for the test, since it helps the algorithm assign as many known values as possible and hence refines existing possibilities.

a) Value assignment implication

We define value assignment implication as a process of determining the value defined or used at various locations of a program because of some other new value assignments. It can be performed backward or forward, i.e., from a *definition* to a *use* or from a *use* to a *definition*. It can also be performed for the variables used in an *EC* when the *EC* must or must not be satisfied. A *conflict* is identified if the implied value is not compatible with the current value.

Here we show an example of *value assignment implication*. In Fig. 2(a), *def(x,1)* and *def(x,3)* are two *actively reaching definitions* of *use(x,6)*. Consider a backward value assignment implication task for *def(x,6)* with a value of 2 is performed. We identify *mROA* for statement 6 and its symbolic expression "$x\_out = x + 1$", then we send this expression and "$x\_out = 2$" to solver, which gives us the unique solution of "$x = 1$". Thus, 1 is assigned to *use(x,6)* and a backward value assignment implication task for *use(x,6)* is added to *implication task list* ($\mathcal{I}$). For this new task, we find *use(x,6)* has two *actively reaching definitions*, that means no value should be *implied* and *use(x,6)* is added to *unjustified element list* ($\mathcal{U}$).

b) IVS implication

IVS implication is defined as a process of determining the values of IVS of nodes/edges of a program's CFG as a result of the changes in value of IVS of other nodes/edges. A *conflict* is identified if the implied value is not compatible with the current value. IVS implication can be performed backward or forward, from *node* to *edge* or from *edge* to *node*.

In Fig. 2(b), consider that node 13 is invoked and a backward IVS implication task is processed for this node. At least one of node 13's incoming edges must be invoked but we do not know which. Hence, no value can be assigned and node 13 is added to $\mathcal{U}$. Consider another case that a forward IVS implication is processed for the invoked edge 9-13 and the IVS values of edge 11-13 and node 13 are both unknown. It first assigns *true* to the IVS of node 13 and adds a forward IVS implication task for this node to $\mathcal{I}$. Then, it assigns *false* to the IVS of edge 11-13 and adds a backward IVS implication task for this edge to $\mathcal{I}$.

2) Mutation effect excitation (MEE) subtask

MEE is performed at the location of the mutation for generating *mutation effect*. This is the necessary condition for killing the mutant. We first identify the *mutation effect excitation mROA* (MEEmROA) and its related information at the location of the mutated statement using the methods mentioned above. Since the *mutation effect* must be present at one or more outputs of MEEmROA's ROD, we create MEE subtask for each variable in MEEmROA's ROD.

When MEE subtask is processed, MEEmROA's symbolic expression for the corresponding *ROD variable* is generated for both original program and the mutant. Then we add the constraint that the values defined at the ROD in original program and the mutant are different. This ensures the *excitation* of the *mutation effect*. These constraints, along with all known values, are then sent to the solver to generate possible solutions.

If a solution is found, *mutation effect* can be *excited*. Then, we assign values to the variables in MEEmROA's RID and ROD. In addition, MEEmROA must be *invoked* to excite the *mutation effect*, thus, we assign *true* to the IVS of MEEmROA's entry node and assign *true* to the IVS of MEEmROA's exit node. Finally, we create implication tasks for all newly assigned values.

3) Mutation effect propagation (MEP) subtask

MEP is performed to propagate *mutation effect* until it is exposed at program's POD. We first identify the *mutation effect propagation mROA* (MEPmROA) at the location of the statement where *mutation effect* presents (at a use of a variable). The remaining process of handling MEP subtask is similar to the MEE subtask, so we do not describe here due to space limit.

4) Justification (JUST) subtask

When any *mutation effect* is present at one or more of program's PODs, we need to *justify* all *unjustified* elements, i.e., $\mathcal{U}$ must be empty. Several different cases need to be considered, including *unjustified* IVS of a *node/edge*, *unjustified* value assignment of a *definition*/*use*, and *unjustified* EC. In Fig. 2(a), if node 6 is invoked, and both of edge 1-6 and edge 3-6 are with unknown IVS, we can justify the *IVS* of node 6 by invoking edge 1-6 and not invoking edge 3-6. Then, new backward IVS implication tasks are added to $\mathcal{I}$ for all edges with changed IVS.

5) Initialization

At the beginning of SW D-algorithm, we need to create an *implication task list* ($\mathcal{I}$), a *subtask stack*, and an *unjustified element list* ($\mathcal{U}$). Also, any known IVS and value assignment need to be assigned, including the value *true* to the IVS of the entry/exit node, and all concrete definitions of variables. The corresponding implications tasks are then being added to $\mathcal{I}$.

6) Backtrack and search space

SW D-algorithm requires *backtrack* when a *conflict* is identified, all temporary information of the program must be erased, and the previous state must be restored.

In HW digital circuit, the logic function of a gate can be represented using a *truth table*, which has limited number of entries. In SW program, we use constraint expressions to describe the behavior of an mROA. In most cases, it is difficult to list all solutions of the constraints because individual variables in SW can have extremely large value spaces (e.g., a variable with integer value). Therefore, it is crucial to set up a limit on how many solutions are explored for a constraint expression before it *backtracks*, we call this limit as *backtrack limit*.

In practice, a small backtrack limit (3 to 5) is usually sufficient to achieving high mutation score, although the search space for SW D-algorithm is huge. This is because in most cases, there is potentially a large set of possible tests to kill a mutant.

7) A running example

Here we show an example to illustrate SW D-algorithm. Fig. 4 shows the search tree for this process (VA means "value assignment"). Consider the mutant shown in Fig. 3, after we initialize the whole program and perform implication, IVS values of node 0, 1, and 2 are all *true*, *def(x,1)*=0/0, *use(x,3)*=0/0, and *def(x,3)*=1/1. Then, *MEEmROA* is identified and processed, we assign *use(b)*=0/0, *use(c)*=0/0, *use(x)*=0/0, and *def(x)*=1/0 to variables in *MEEmROA's RID* and *ROD*. After implication

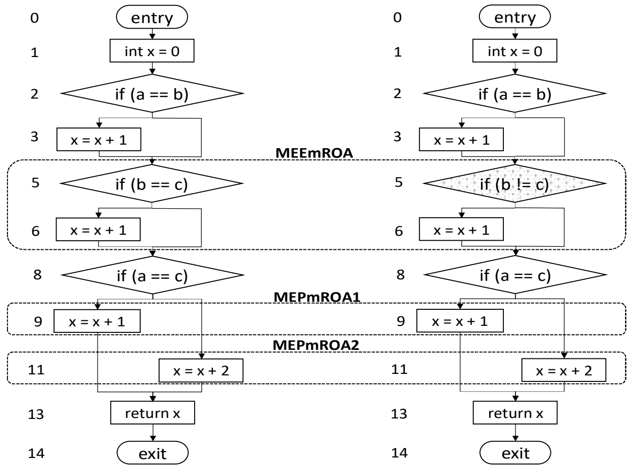

Fig. 3. CFG of the original program and its mutant (at statement 5)

procedure, node 8 is invoked, *use(x,9)*=1/0, *def(x,9)* = 2/1, *use(x,11)*=1/0 and *def(x,11)*=3/2. Therefore, two MEP subtasks, through *use(x,9)* or *use(x,11)* are created. We first try *MEPmROA1* for *use(x,9)*. During implication, a *conflict* is found at *use(x)* of *MEEmROA's RID*, where the existing value is 0 and the newly implied value is 1. Therefore, the algorithm *backtracks* to the last option, which is to propagate *mutation effect* via *MEPmROA2* of *use(x,11)*, and the algorithm state is *recovered*. *MEPmROA2* is then processed and all required implication tasks are performed afterwards *without conflict*, and the *mutation effect* is present at line 13 (the returned value) which belongs to program's *POD*. Although, two *unjustified* elements, *EC* of edge 8-11 and *IVS* of node 5, are still left in $\mathcal{U}$. So, we push these *justification* subtasks to the *subtask stack* and process them. We first process *justification* for the *unjustified EC* of edge 8-11. Thus, we have a new value assignment of *use(a,8)*=1/1. Implication is performed afterwards and $\mathcal{U}$ is now empty. We generate a test: "*a=1*", "*b=0*", and "*c=0*" at program's *PID*, and the *mutation effect* at program's *POD* is "*def(retVal,13)*=3/2".

C. SW ATG (Automatic Test Generation) system

We propose SW ATG system to perform a *2-pass test generation*. In the *first pass* we use random test generation, and in the *second pass* we use the SW D-algorithm. In the third pass, inspired by HW testing, we perform *reverse order simulation* to eliminate redundant tests.

The first pass cheaply kills many mutants and reduces the run-time complexity of our deterministic algorithm. Reverse order simulation makes test set compact by eliminating many random tests, keeping only tests that kill mutants not killed by deterministic tests simulated earlier. All steps of our SW ATG system are outlined below:

i. Given a program, generate mutants by applying each mutation operator; store all mutants in the mutant list, $\mathcal{M}$.

ii. Perform random test generation to obtain *r* tests, simulate all mutants in $\mathcal{M}$ for these *r* tests. Eliminate all mutants killed from $\mathcal{M}$ and only keep tests that kill at least one mutant.

iii. While $\mathcal{M}$ is not empty

- SW D-algorithm: Select a mutant from $\mathcal{M}$, use our SW ATG to generate a deterministic test.
- Mutant dropping: Perform *mutation simulation* (*mutation testing*) by applying the newly generated test, eliminate any mutant killed from $\mathcal{M}$.

iv. Perform reverse order simulation, eliminate redundant tests to obtain a compact test set.

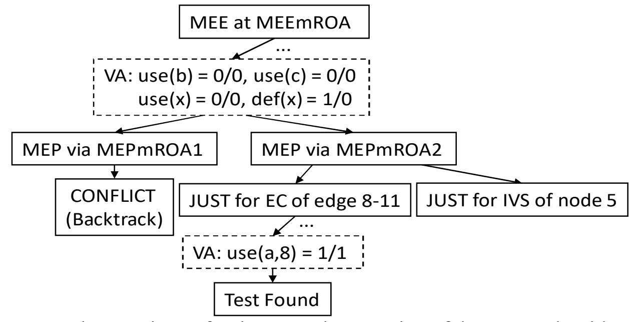

Fig. 4. The search tree for the example execution of the SW D-algorithm

V. Implementation

Our tools are developed in Java, the tools can generate tests for Java programs with primitive data types (this is also the limitation of all existing constraint-based TG tools, we are now working to expand the scope to all data types). The symbolic analysis engine is based on JPF-SE [16], while we make many changes to accommodate all specific requirements on our new SW D-algorithm. The constraint solver used is Z3 [17], which is a commonly used powerful theorem prover. In addition, we use muJava [18] to generate/simulate mutants and use the method we proposed in [11] to eliminate equivalent mutants beforehand. *We note that our SW D-algorithm works on Java bytecode, while here we use Java source code to illustrate our algorithm.*

VI. Results

We test our SW D-algorithm and SW ATG system on programs from Ammann and Offutt's text [9], Demillo and Offutt's paper [7], and from leetcode.com. These programs' sizes range from 50 to 200 lines of code, which are typical sizes of the program modules for unit testing. Also, "triang" is a widely used testbench appears in many papers on mutation testing.

The baseline approach uses global symbolic analysis. It generates constraint expressions for original program and the mutant and the constraint that at least one program output has different values across the original program and the mutant. Any solution satisfies these constraints is a test for the target mutant.

We compare our method with EvoSuite [20], which is one of the best state-of-the-art SW test generation tools and won the SBST 2017 tool competition. In our experiments, two configurations for EvoSuite are used. Default configuration uses balanced weight of different metrics (e.g., line coverage, branch coverage, mutant, etc.) in the fitness function. Strong mutation configuration only includes strong mutation metrics in its fitness function. Also, we use two search time budgets (60s and 120s) for each configuration.

Given a program under test, we use muJava [18] to generate all its mutants. Then we compare our methods with the baseline method and EvoSuite in three aspects: *mutation score,* number of tests generated, and runtime (in seconds). We first use the method described in another paper [11] from us to eliminate all equivalent mutants, which provides the "# of non-equivalent mutants" in Table II. For all non-equivalent mutants, we use the baseline approach, EvoSuite with different configurations and search time budgets, as well as our SW D-Algorithm and SW ATG to generate tests and obtain the statistics of interest.

In Table II, although the baseline approach provides the best quality tests (100% in all cases), it is very time consuming compared to the other two methods, and it generates too many test cases. EvoSuite with default configuration obtains good

TABLE II. COMPARISON AMONG DIFFERENT APPROACHES

Program name	Program size	# of mutants	# of non-equivalent mutants	Baseline approach			EvoSuite - default 60s			EvoSuite - default 120s			EvoSuite - strong mutant 60s			EvoSuite - strong mutant 120s			SW D-Algorithm			SW ATG		
				score	size	time	score	size	Time	score	size	time	score	size	time	score	size	time	score	size	time	score	size	time
isPalindrome	97	230	161	100	161	732	64	6	60	63	6	120	31	5	60	40	5	120	84	10	91	88	6	72
findLastEx	52	102	69	100	69	218	86	4	60	86	4	120	84	4	60	84	4	120	94	8	7	95	6	6
countPositiveEx	68	166	104	100	104	749	92	3	60	91	5	120	93	4	60	99	5	120	95	17	9	100	9	6
triang	166	315	265	100	265	960	89	21	60	92	21	120	82	18	60	84	18	120	98	22	74	98	19	114
reachNumber	71	221	126	100	126	395	82	8	60	82	8	120	69	7	60	70	7	120	83	12	57	87	10	51
numZeroEx	69	140	98	100	98	653	97	4	60	98	4	120	98	5	60	98	5	120	95	12	22	95	8	7
switchButton	101	127	94	100	94	541	80	5	60	80	5	120	70	3	60	74	3	120	85	11	135	86	5	102
reverseInteger	128	249	166	100	166	752	85	4	60	85	3	120	32	3	60	9	2	120	95	8	110	99	4	7

scores (i.e., coverage), but increasing the search time doesn't improve score. Surprisingly, sometimes the scores decrease. This could be due to the local maxima problem of the search-based algorithm. EvoSuite with strong mutant configuration doesn't perform as well as the default setting. One possible reason is that by using default setting, mutant coverage benefits from other components in the fitness function. For example, many mutants can be killed if the mutated line is covered. Thus, the line coverage metric automatically helps mutant coverage.

Our SW ATG system obtains better mutation scores compared to EvoSuite for 7 out of 8 programs. Also, we achieve a significant speed-up by using our SW ATG system, compared to the baseline approach and EvoSuite in most cases. Also, our SW ATG system decreases the number of tests, compared to SW D-Algorithm. This is very important given that a compact test data is crucial in SW testing, since testers need to manually check the expected outputs for each test (test oracle problem [21]).

Overall, our SW ATG system provides a higher mutation score with a shorter run-time and generates compact test. In summary, our new method is more scalable than the baseline and effective when high quality tests are required.

VII. CONCLUSION

In this paper, we propose a new method called SW D-algorithm for mutation-oriented test generation for software unit testing. We define new concepts and methods to capture unique characteristics of SW and extend HW D-algorithm for SW test generation. Specifically, we define new concepts including *IVS*, *active DU chains*, *mROA*, and *mutation effect*. And we create a series of innovative approaches and procedures for SW D-algorithm. In addition, we combine SW D-algorithm, random test generation, and reverse order simulation to build a more efficient SW ATG system. Compared with previous constraint-based mutant-oriented test generation methods, our approach is more scalable and effective. Our tools also outperform EvoSuite, the state-of-the-art test generation tool, in terms of test quality.

Our ongoing research includes implementing multiple heuristics to make our approach more scalable, expanding our method for handling programs with more complex data types and more complex structures. The details can be found in [15].

REFERENCES

[1] J.P. Roth, "Diagnosis of Automata Failures: A Calculus and a Method," *IBM Journal of Research and Development*, vol.10, no.4, pp.278,291, July 1966.

[2] P. Goel and B.C. Rosales, "PODEM-X: An Automatic Test Generation System for VLSI Logic Structures," *Design Automation, 1981. 18th Conference on* , vol., no., pp.260,268, 29-1 June 1981.

[3] F. Del Frate, P. Garg, A. P. Mathur and A. Pasquini, "On the correlation between code coverage and software reliability," Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on, Toulouse, 1995, pp. 124-132.

[4] L. Inozemtseva, and R. Holmes. "Coverage is not strongly correlated with test suite effectiveness." Proceedings of the 36th International Conference on Software Engineering. ACM, 2014.

[5] R.A. Demillo, R.J. Lipton and F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, vol.11, no.4, pp.34,41, April 1978.

[6] Just, René, et al. "Are mutants a valid substitute for real faults in software testing?" *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014.

[7] R.A. Demillo and A.J. Offutt, "Constraint-based automatic test data generation," *Software Engineering, IEEE Transactions on*, vol.17, no.9, pp.900,910, Sep 1991.

[8] I. Ghosh and M. Fujita, "Automatic test pattern generation for functional RTL circuits using assignment decision diagrams," Proceedings 37th Design Automation Conference, Los Angeles, CA, USA, 2000, pp. 43-48.

[9] P. Ammann and J. Offutt, *Introduction to Software Testing.* Cambridge Univ. Press, 2008.

[10] L. Bottaci, "A Genetic Algorithm Fitness Function for Mutation Testing", in Proceedings of the Software Engineering using Metaheuristic innovative Algorithms workshop, April 2001, pp. 3-7.

[11] J. Zhang and S. K. Gupta, "Using hardware testing approaches to improve software testing: Undetectable mutant identification," 2016 IEEE 34th VLSI Test Symposium (VTS), Las Vegas, NV, 2016, pp. 1-6.

[12] King, C. James, "Symbolic execution and program testing." *Communications of the ACM* 19, no. 7, pp. 385-394, 1976.

[13] T. Lengauer and R.E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 1, pp. 121-141, 1979.

[14] ORACLE JAVA Documentation, Lesson: Basic I/O, Available: https://docs.oracle.com/javase/tutorial/essential/io/index.html.

[15] J. Zhang, "Software Automatic Test Generation System," Ph.D. Dissertation Proposal, available: https://bit.ly/2DKasWb

[16] S. Anand, C.S. Păsăreanu and W. Visser, "JPF–SE: A symbolic execution extension to java pathfinder," in Tools and Algorithms for the Construction and Analysis of Systems, Springer, pp. 134-138, 2007.

[17] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver." International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 2008.

[18] Y. Ma, J. Offutt and Y.R. Kwon, "MuJava: an automated class mutation system," *Software Testing, Verification and Reliability*, vol. 15, pp. 97-133, 2005.

[19] Y. Serrestou, V. Beroulle and C. Robach, "Functional Verification of RTL Designs driven by Mutation Testing metrics," *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, Lubeck, 2007, pp. 222-227.

[20] EvoSuite, available: http://www.evosuite.org/

[21] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz and S. Yoo, "The Oracle Problem in Software Testing: A Survey," in *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507-525, 1 May 2015.

[22] S. Krishnamoorthy, M. S. Hsiao and L. Lingappan, "Tackling the Path Explosion Problem in Symbolic Execution-Driven Test Generation for Programs," *2010 19th IEEE Asian Test Symposium*, Shanghai, 2010, pp. 59-64.