Path-Oriented Test Data Generation Using Symbolic Execution and Constraint Solving Techniques*

Jian Zhang, Chen Xu† and Xiaoliang Wang
Laboratory of Computer Science
Institute of Software
Chinese Academy of Sciences
Beijing 100080, China
Email: zj@ios.ac.cn

Abstract

Automatic test data generation is a challenging task in software engineering research. This paper studies a path-oriented approach to the problem, which is based on the combination of symbolic execution and constraint solving. Methods for representing expressions and path conditions are discussed. An implemented toolkit is described with some examples. The toolkit transforms an input program (possibly embedded with assertions) to an extended finite state machine and then performs depth-first or breadth-first search on it. The goal is to find values for input variables such that a terminal state can be reached. If successful, input test data are found (which might reveal a bug in the program).

1. Introduction

Testing is an important activity in software development. But the state of the art leaves much to be desired. Depending on the time and scale of testing, we may distinguish between unit testing, integration testing, system testing and user acceptance testing. On the other hand, we also divide testing approaches into two categories: white-box testing which is based on the source code, and black-box testing which is based on the specification. This paper focuses on the white-box approach to *unit testing*, and the main goal is to find appropriate test data automatically. Such input data will drive the program to execute along selected paths, while satisfying all the assertions (conditional expressions) in each path. If we put the negation of a correctness property at the end of the program, and input data are found such that the program reaches the terminal state, a bug is revealed.

An important class of testing methods is called path-oriented testing, that is, analyzing various paths of the target program. Traditionally, many path-oriented methods are automatic, but they neglect useful semantic information such as the values of the variables. A typical test case generation method first identifies a set of paths in the program's flow graph, which covers all branches or all statements. Then, it tries to find input test data such that every selected path is executed. However, usually it turns out that a large portion of the paths can not be executed [28].

Our approach is mainly based on symbolic execution, i.e., executing each path with symbolic values for variables. All semantic information is employed in the anlaysis, and the resulting test cases are all executable. Although symbolic execution is a powerful analysis method, it has not been used much in practice. To quote Gupta *et al.* (page 231 of [17]), it "requires complex algebraic manipulations and has difficulty in handling arrays and pointer references." Offutt *et al.* (page 168 of [25]) also note that symbolic execution "has several practical problems, including aliasing, solving for indeterminate loops, and the size of the symbolic expressions." In this paper, we attempt to tackle some of these problems.

The paper is organized as follows. In the next section, we recall some basic concepts and notations which will be used later. In Section 3, we elaborate on our method for test data generation. We shall discribe the basic ideas of symbolic execution, constraint solving and present the main algorithms and data structures. Several implemented tools are described in Section 4, together with some examples. Finally, we compare our approach with other related works, and discuss possi-

* Supported by the National Science Fund for Distinguished Young Scholars (No. 60125207).
† This author is currently studying at Yale University.

Proceedings of the Second International Conference on Software Engineering and Formal Methods (SEFM'04)
0-7695-2222-X/04 $ 20.00 IEEE

ble improvements in the future.

2. Basic Concepts and Definitions

In this section, we describe some basic concepts which will be used later. We also give some definitions, notations and assumptions.

2.1. A Small Programming Language

We use C-like syntax to describe both our analysis algorithms and the target programs that are analyzed. Thus the symbols '`=`' and '`==`' stand for assignment and equality, respectively; "a `!=` b" means that a and b are not equal. The logical operators NOT, OR, AND are denoted by '`!`', '`||`', '`&&`', respectively. An array `a` of size `N` consists of the elements `a[0]`, `a[1]`, ..., `a[N-1]`.

The input program may have the above logical operators and arithmetic operators like '+' and '-', but there can not be non-linear operators like multiplication. An assertion is given by the statement `assert(conditional_expression)`.

Rather than presenting a detailed syntax for the language, we shall give some examples later, and show the exact texts that are accepted by our tools.

2.2. Extended Finite State Machines

In the literature, the target program is typically represented in some graphical form such as a control flow graph or a data flow graph. Some information is abstracted away in these graphs. In our case, we use an extended finite state machine (EFSM), which encodes almost all the information in the program.

We define an EFSM to be the triple $\langle S, V, T\rangle$ where

- S is the set of states,
- V is a set of variables, and
- T is a set of transitions.

We assume that the set S contains one *initial* state, denoted by s_0, and one or more *terminal* states which represent the end of a program's execution.

Each transition t is a tuple $\langle cs_t, ns_t, pr_t, ac_t\rangle$, where

- cs_t and ns_t denote the current state and the next state of the transition, respectively;
- pr_t is the predicate, i.e., a set of conditional expressions or assertions about the current values of the variables, and
- ac_t is the action, i.e., a set of input statements or assignment statements which give values to the variables.

transition	predicate	action
t_0		`scanf("%d",&m);` `scanf("%d",&n);`
t_1	`(m>1)&&(n>1)`	`x = m; y = n;`
t_2	`(x==y)`	`gcd = x;`
t_3	`(x!=y)`	
t_4	`(x<=y)`	`y = y-x;`
t_5	`(x>y)`	`x = x-y;`

Table 1. Definitions of the transitions in Fig. 1

Note that we do not consider output statements in this paper, since we are mostly interested in the functional aspects of the program.

The EFSM is "executed" in the following way. Initially it is at state s_0. Later, it can perform a set of "movements", until it reaches some terminal state. Each movement is like this. Suppose the current state is s and the values of the variables are denoted by the vector $\vec{v}$. The EFSM may take the transition $t = \langle s, n, p, a\rangle$ and move to the state n, if $p(\vec{v})$ is TRUE. While taking this transition, it updates the values of the variables by performing the action a.

An EFSM is *deterministic* if for any pair of transitions t_1 and t_2 originating from the same state s, the predicate of t_1 and that of t_2 can not be satisfied simultaneously. Otherwise, it is *non-deterministic*.

Example 1. The following program uses Euclid's algorithm to compute the greatest common divisor of two positive integers.

```
void main()
{
  int m, n;
  int x, y;
  int  gcd;

  scanf("%d", &m);
  scanf("%d", &n);
  assert((m > 1) && (n > 1));
  x = m;
  y = n;
  while (x != y) {
    if (x > y)
        x = x-y;
    else y = y-x;
  }
  gcd = x;
}
```

The EFSM for this program is given in Fig. 1. Here state 1 is the initial state and state 2 is the terminal state. The predicate and action for each transition is given in Table 1.

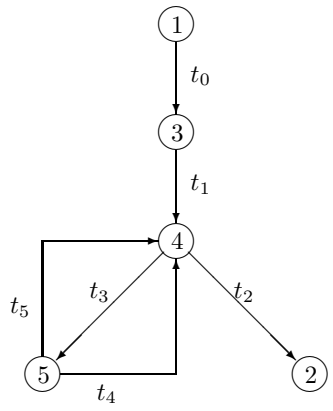

Figure 1. EFSM for the GCD Program

We have already implemented a tool for obtaining a deterministic EFSM from a program written in a subset of C [29]. After the transformation, an assertion in the program becomes the predicate of some transition in the EFSM. Most programs we tested are within 100 lines of code, and the execution time is typically a fraction of a second.

We can also translate a deterministic EFSM back to a C program. This is straightforward. For each state i, we introduce a label L_i. Then the transitions correspond to (conditional) statements.

2.3. Program Paths and Their Feasibility

An EFSM may be treated as a directed graph, and a path can be defined as in graph theory. Here we require that each path begins with the initial state. The *length* of a path is the number of transitions in it. A path is *executable* (or *feasible*) if there are input data such that the program is executed along that path.

For the program in Example 1, one executable path consists of these states and transitions: $1 \stackrel{t_0}{\rightarrow} 3 \stackrel{t_1}{\rightarrow} 4 \stackrel{t_2}{\rightarrow} 2$. The following input data serves the purpose: $m = n = 2$.

In general, there are many or usually an infinite number of paths. However, some paths may not be executable. It is reported in [28] that for some program, only 1.4 percent of the paths (up to a certain length) are executable. This causes a severe problem to path-oriented testing.

The path feasibility problem, in the most general form, is undecidable. However, with certain restrictions, it can be decidable. We have implemented an automatic tool called PAT [32] which determines the feasibility of a program path, under the condition that the path does not involve non-linear arithmetic operators.

3. Symbolic Execution and Constraint Solving

By test data generation we mean finding a set of test cases for a given program such that some criterion is satisfied. The most commonly used criteria include that every statement is executed and that every branch of the flow graph is traversed.

Korel and Al-Yami [21] suggest finding test cases which violate given assertions. They propose to translate the assertions to code fragments, and then use existing methods of automated test data generation for white-box testing (e.g., the chaining approach). We follow this methodology, which is usually more challenging and more meaningful than other criteria.

3.1. Symbolic Execution

Symbolic execution [3, 8, 20] is a useful technique for verification and testing, which has been studied by various researchers [5, 9, 14, 19] during the past 30 years. Its basic idea is very simple, namely, to "execute" a program using symbolic values for variables.

The actual execution of a program requires concrete values (e.g., $x == 3$, $y == 10$) as the input data. In contrast, symbolic execution uses symbols as values of variables. For instance, $x == a_0$, $y == b_0$. The execution usually involves operations on complex expressions. Suppose there is a statement like $z = x + 2y$. With ordinary program execution, the result of z will be a concrete value like 23. But with symbolic execution, the result is typically some symbolic expression like $a_0 + 2b_0$. After many steps of execution, a variable's value may become a very complex expression.

3.2. Constraint Solving

A key problem with symbolic execution is to decide whether a set of constraints is satisfiable. Continuing with the above example, suppose there is a statement `if (z < x) S1 else S2`. Then we need to know whether $x + 2y < x$ holds in the current context. If there is a precondition saying that y must be positive, the condition will not be satisfiable. Thus the statement `S2` will be executed and `S1` will not. Otherwise, we might need to execute both `S1` and `S2`.

To decide the satisfiability of conditions, we can use constraint solving techniques which have been studied by many researchers in the artificial intelligence (AI)

and operations research (OR) communities. An important problem paradigm in AI is the so-called constraint satisfaction problem (CSP) [22]. Informally speaking, a CSP consists of a set of variables, each of which may take a value from some domain. In addition, there are some constraints defined on the variables. Solving a CSP means finding a value for each variable, such that all the constraints hold.

Obviously, CSP represents a very general class of problems (e.g., graph coloring, satisfying formulas in the propositional logic). The more expressive the constraint language is, the more difficult it is to solve the constraints. For example, if multiplication between integer variables is allowed, the problem is undecidable.

More often than not, people focus on some special forms of constraints, such as systems of linear equations or inequalities. For the purpose of test generation, we have designed and implemented a tool (called BoNuS [32]) for solving constraints of more general forms. The solver can accept constraints written in a very natural form, involving variables of various types (including integer, float and enumeration). The constraints can have logical operators (like AND, OR) and arithmetic operators (like addition, subtraction, but not multiplication and other non-linear operators).

3.3. Main Algorithm

Symbolic execution can be done in two directions: from the initial state to the terminal states, or vice versa. We adopt the former one, which has certain advantages [3]. Our main algorithm, which is basically a depth-first search (DFS) procedure, is given in Fig. 2. It tries to find appropriate initial values of the variables such that the EFSM may reach a terminal state. This is achieved by repeatedly extending or shortening feasible paths.

In Fig. 2, PathCond refers to the path condition, which represents the set of input data that can drive the program to the current state along the current path. Besides the path condition, we also need to record and update every variable's value (which is a symbolic arithmetic expression).

The algorithm has been implemented in a tool called EFAT (EFsm Analyzer and Test generator) [33]. The user may write assertions in the input program, and then asks EFAT to search for a counterexample. Suppose the program has several properties: Pty_1, Pty_2, ... The user may run our tool several times, each time with the input

$PreCond$ Program $PostCond_i$.

Here $PreCond$ is the precondition, and $PostCond_i$ is the negation of Pty_i. If in any run of EFAT, input data are found which satisfy the assertions, an error is detected. EFAT may also be used to analyze path feasibility, since a single path is a special form of an EFSM.

```
void DFS()
{
    CurPath = empty;
    PathCond = empty;
    CurState = s0;    // initial state
    while (true) {
        if CurState is a terminal state {
            solve the constraints PathCond and
                output the variables' values;
            return;
        }
        select a transition tr which leaves
            CurState and whose predicate can
            be satisfied under PathCond;
        if the transition tr does not exist {
            if CurState is s0
                return;    // Failure
            else backtrack and modify
                    CurState, PathCond and CurPath;
        }
        else {
            add tr's predicate to PathCond;
            perform tr's action;
            add tr to the end of CurPath;
            change CurState to tr's next state;
        }
    }
}
```

Figure 2. Test Data Generation Algorithm

It should be noted that a naive depth-first search procedure such as the algorithm in Fig. 2 may not terminate even if the target program terminates on any input data. Consider the `gcd` program in Example 1. Suppose the initial values are: $m = 2,\ n = 2k\ (k > 1)$. Then the program will take the loop `t3-t4` for $(k - 1)$ times. In other words, if the test data generation algorithm keeps trying this loop, it will always be able to find suitable values for the variables.

This problem may be avoided if we impose some restriction on the search procedure. For example, each state can be visited for at most MT times.

3.4. Data Structures

During the search, we need to store and process many symbolic expressions and constraints. They are given in terms of the initial values of the input variables. As mentioned earlier, they can become very complex during symbolic execution. To circumvent this problem, it is better to simplify them.

A simple example of constraints is $a_0 + 2b_0 < c_0$. We call it an *arithmetic constraint*, which is a comparison between two arithmetic expressions. There can also be more complicated constraints, which are combinations of arithmetic constraints using logical operators.

First let us look at the arithmetic constraints. Since there are no non-linear operators, we normalize every arithmetic constraint into the form *expr op* 0, where *expr* is a linear polynomial and *op* is a comparison symbol (e.g., '$>$', '$\leq$').

Now we turn to more complex constraints involving logical operators. We define a *literal* to be either an arithmetic constraint, a Boolean variable or its negation. A *conjunctive clause* is a list of literals connected by logical ANDs, and a *disjunctive clause* is a list of literals connected by logical ORs. A clause is called a *unit clause* if it has only one literal. A constraint is in the *Disjunctive Normal Form (DNF)*, if it is a list of conjunctive clauses connected by logical ORs. Similarly, a constraint is in the *Conjunctive Normal Form (CNF)*, if it is a list of disjunctive clauses connected by logical ANDs.

For example, here is a disjunctive clause: $(a_0+2b_0 < c_0)$ `||` $(a_0 > b_0)$. And the following is a constraint in DNF: $(p$`&&`$q)$`||`$(a_0 > 9)$.

We represent complex constraints either in CNF or in DNF. Note that we use both CNF and DNF, so that common logical operations on the constraints can be easily implemented. For example, to negate a CNF, we can easily obtain the result as a DNF. In contrast to our scheme, all constraints are kept in DNFs in [25].

In our implementation, each constraint is stored as a list of clauses. There is a type indicator to distinguish between CNF and DNF. As special cases, a constraint can be `TRUE` or `FALSE`. A `TRUE` constraint can be deleted immediately. On the other hand, if there is a `FALSE` constraint, the current path can not be extended and backtracking occurs.

To perform the logical operations on the constraints, we have implemented several routines including

- `And_CNF_CNF`
- `And_DNF_DNF`
- `Or_CNF_CNF`
- `Or_DNF_DNF`
- `And_CNF_DNF`
- `Or_CNF_DNF`

Our experiences show that the most commonly used routines are `And_CNF_CNF` and `Or_DNF_DNF`, whose computational costs are quite low. `Or_DNF_DNF` is called more often than `And_CNF_CNF`, while the other routines are rarely called.

4. Tools and Examples

We have implemented a number of tools which form a toolkit for finding errors in small programs. They include the following:

- GENESIS (GENerator of Extended finite State machIneS), whose input is a program written in a subset of C. Details of the method are given in [29].
- EFAT (EFsm Analyzer and Test generator). Its input is an EFSM and its output is a vector of initial values for variables, if that exists, such that the EFSM reaches a terminal state.
- PATEN (PATh ENumerator). Given an EFSM as input, it generates paths systematically, from shorter ones to longer ones.

Collectively the tools can assist the programmer or test engineer in various ways, such as deciding the feasibility of a particular path, finding potential bugs and so on.

Example 3. There is a wrong version of the bubble sort algorithm. First let us check the property that the result is a non-decreasing array. We negate the property and use it as an assertion (actually a postcondition). The input program is like this:

```
int a[5];

void bubble()
{
   // code omitted
}

void main()
{
  scanf("%d", &a[0]);
  scanf("%d", &a[1]);
  scanf("%d", &a[2]);
  scanf("%d", &a[3]);
  scanf("%d", &a[4]);
  bubble();
  assert((a[0]>a[1])||(a[1]>a[2])
       ||(a[2]>a[3])||(a[3]>a[4]));
}
```

EFAT fails to find any input data. Thus the aforementioned property holds. However, this does not mean that the program is correct. We may add another assertion, which is the negation of the property that an element in the initial array should also appear in the result array. To check this, we change the `main` function as follows.

```
void main()
{
  int t0, t1, t2, t3, t4;
  scanf("%d", &a[0]);
  scanf("%d", &a[1]);
  scanf("%d", &a[2]);
  scanf("%d", &a[3]);
  scanf("%d", &a[4]);
  t0 = a[0];
  t1 = a[1]; t2 = a[2];
  t3 = a[3]; t4 = a[4];
  bubble();
  assert((t2!=a[0]) && (t2!=a[1])
      && (t2!=a[2]) && (t2!=a[3])
      && (t2!=a[4]));
}
```

Now EFAT gives us an input array which satisfies the postcondition: `{1, 0, 0, 0, 0}`. This serves as a counter-example showing the incorrectness of the program. Thus a good assertion plays an important role in identifying errors in a program.

Our tools may also be combined with other techniques and tools. For example, one could use the method described in [2] to obtain a set of paths, and then apply EFAT to decide the feasibility of each path.

We have tested EFAT on several programs [33], in addition to bubble-sort. Some of them are listed below.

- `qsort`: quicksort
- `kmp`: KMP string matching algorithm
- `find`: a program which rearranges an array such that all elements on the left are less than or equal to a given integer f, and all elements on the right are greater than or equal to f.
- `grader`: a program computing the average grades of students, which has about 100 lines of code.

We use a SUN SPARCserver with 1G memory and two 400MHz CPU. The running times (in seconds) of EFAT are given in Table 2. In the table, a problem name followed by "`-wr`" is a wrong version of the program. For each problem, there are two columns showing the number of states and the number of transitions in the corresponding EFSM.

It can be seen from Table 2 that it is usually more difficult to show a program's correctness than to find test data for an incorrect version of the program. The reason is that, with our approach, showing the correctness requires the examination of all paths.

Problem	n_state	n_trans	time
`bsort`	11	12	4.9
`bsort-wr`	11	12	0.1
`qsort`	12	16	3.4
`qsort-wr`	12	16	0.1
`kmp`	16	17	0.6
`kmp-wr`	16	17	0.3
`find`	13	17	7.0
`find-wr`	13	17	0.1
`grader`	17	24	0.7

Table 2. Running Times of EFAT

The current version of EFAT does not support breadth-first search. But we can use EFAT as a path feasibility analyzer and combine it with PATEN. The combined tool repeatedly extends feasible paths.

For a given program, we first generate a set of paths using PATEN, and then for each path, we use EFAT to decide its feasibility.

5. Related Work

Path Feasibility Analysis

As we briefly mentioned in Section 2, determining path feasibility is a key problem in path-oriented program analysis and test data generation. Most previous works either avoid this problem or use approximate methods. For example, in [30], the number of predicates in a path is used as a measure of its feasibility. According to the authors' empirical results, the fewer predicates a path has, the more likely it is feasible. Thus a strategy of test data generation is choosing paths involving a minimum number of predicates.

Bertolino and Marré [2] described an algorithm and various policies for selecting a set of paths, such that the number of infeasible paths is reduced. However, as the authors claimed (page 893 of [2]), their algorithm "cannot completely fulfill the specific objective of deriving executable paths, as it is only based on the analysis of program control flow."

A powerful method for deciding the feasibility of paths is proposed by Gupta *et al.* [17]. It is especially suitable for numerical computation applications. But logical expressions seem to be difficult for the method.

Symbolic Execution

The basic idea of symbolic execution has been known for a long time [3, 8, 20]. However, existing systems have a number of limitations, some of which are mentioned in Section 1. For example, the UNISEX system [19] can only deal with certain expressions, and it is not fully automatic. An important module of UNISEX is the theorem prover, which was "a stub that asks the user to make the reduction" (page 447, [19]). The predicate simplifier "handles expressions involving symbolic and numeric values", but it "cannot deal with the logical operators (and, or, not), nor with conditional expressions" (page 450 of [19]).

More recently, Chen *et al.* [7] also discussed the use of symbolic execution techniques in software testing, but they do not have any tool support. The tool PREfix [4] tries to find errors in programs by path simulation. However, as the authors claimed, their implementation "stops short of full-fledged symbolic computation" (page 781 of [4]), and it is not their goal "to provide a guarantee that all errors of a given type would be found" (page 789 of [4]). To our knowledge, PREfix does not distinguish between different elements of the same array variable. Thus the array expressions `a[i+3]` and `a[0]` are treated as the same thing.

Constraint Representation and Solving

Quite a few researchers have used constraint-based techniques to solve path conditions and to generate test data. But most works can only handle constraints of certain forms. In the tool SELECT [3], algorithms for solving systems of linear equalities and inequalities were implemented. Linear programming was also used in [11]. Gotlieb *et al.* [16] used a constraint solver which can only solve constraints over integers.

A toolkit called Godzilla was described in [13]. It includes a path analyzer, a constraint generator and a constraint satisfier. But the constraints are quite simple, and all array elements are regarded as the same (if the array name is the same). Recently, Offutt *et al.* [25] extended that work significantly, and proposed a new technique called dynamic domain reduction (DDR). It asks the user to supply an initial value domain for each variable, and then uses domain splitting to get the desired test data. It seems that the initial domains are quite small, e.g., [-10, 10]. But in our case, the domains are arbitrary. Moreover, DDR "has only been applied to numeric software", because in their tool, "the expression handling is limited to expressions that use numeric operators" (page 187 of [25]).

Optimization methods like simulated annealing [27] and genetic algorithms [26] have been used to generate test data. These approaches are automatic, and effective in many cases. However, all of them use incomplete constraint solving techniques. That means, a solution may not be found even when there is one. According to Michael *et al.* (page 1104 of [23]), their technique "seems inadequate when the condition contains Boolean variables or enumerated types". It appears that our approach is complementary to optimization methods.

Formal Verification

Conventional formal verification techniques are typically heavy-weight, in the sense that they can prove many correctness properties written in an expressive specification language, yet the proofs may be very difficult to find. In recent years, however, some researchers are beginning to adapt those methods for automatic analysis of programs. For example, abstract interpretation [10] can be applied for such purposes. In addition, tools like ESC/Java [15] also use verification technology. ESC/Java can deal with realistic Java programs, but it only tries to find certain bugs like array bound errors. Moreover, the analysis is neither sound nor complete, and the user has to supply invariants.

Model checking techniques are quite successful on the verification of hardware designs, especially those that can be described by propositional formulas. Recently, there have been some attempts to apply such techniques to verify software requirements/designs or even the source code.

Chan *et al.* [6] combined a model checker with a constraint solver to check formal software specifications. The solver used there can solve non-linear constraints, while we assume that all the primitive constraints are linear. On the other hand, the input language accepted by our tool is very close to a conventional programming language, but in [6], the language is still a fairly abstract one.

In general, to verify software modules on a source code level, one need to use various abstraction methods to get an abstract model which is accepted by a model checker. Ball *et al.* [1] described an algorithm for constructing a predicate abstraction of programs written in C. However, the user has to supply suitable predicates like `(prev->val > v)` which are treated as Boolean variables.

The partial verification tool ESP [12] tries to check that a program satisfies a given temporal safety property. It has been used to verify the file I/O behavior

of a C compiler. ESP works on finite state machines (FSMs), while our tool EFAT works on EFSMs.

6. Concluding Remarks

We have presented an approach to automated test data generation, which is based on symbolic execution and constraint solving. Our focus is on unit testing and the correctness of key algorithms. In this paper, we describe a toolkit for test data generation and partial verification, using some examples.

There are many tools for program analysis and test generation. They may be roughly divided into two categories. The tools in the first category can analyze large programs, but with less accuracy, because they omit much information in the analysis. A typical example is PREfix [4], which is quite successful. Our tools fall in the second category, which produce more accurate results. However, many other tools in this category are either semi-automatic or restrict the forms of expressions and path conditions. The most common restriction is excluding logical expressions. We think that the ability of handling logical operators is very important. Without this, many programs can not be analyzed.

Aiming at small programs, our toolkit is highly automatic and also quite efficient. When EFAT terminates and finds a vector of initial values, it indeed reveals a bug, if the negation of the correctness property is attached at the end of the program.

However, two issues may arise. The first one is whether it is useful to analyze small programs. We believe it is necessary, as unit testing is mainly the responsibility of programmers and automatic tools should be helpful. Moreover, even for small programs or modules, it is difficult to ensure their correctness. As reported in [24], implementations of UNIX utilities contain many bugs. We may also deal with larger programs if we adopt modular interprocedural analysis as ESC/Java [15] does.

The second issue is about combinatorial explosion. Our analysis involves NP-hard problems such as constraint solving. However, experiences show that for many NP-hard problems, only certain problem instances are difficult to solve [18]. Furthermore, since our target programs are small, the combinatorial search may not be too expensive.

In the future, we are going to study other formalisms for representing path conditions, and search heuristics/strategies for test data generation like that of [2]. The efficiency of the constraint solver should be improved. We may also use a more advanced constraint programming system or decision procedure. It is also worthwhile to carry out more case studies.

The input language of our tools allows many common algorithms to be analyzed. However, it is still not expressive enough. Recently, we implemented a simple tool for analyzing program paths which have pointer variables [31]. In the future, we shall extend the input language further. It is also interesting to study specification languages better than assertions. For example, the Java Modeling Language (JML) can be used to specify the behavior of Java modules. Currently, some analysis and testing tools (like JMLUnit and ESC/Java 2) use JML as the specification language. For more details about JML, see
`http://www.cs.iastate.edu/~leavens/JML/`.

Acknowledgements

We are grateful to the anonymous reviewers for their detailed comments and suggestions.

References

[1] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Proc. of the ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation (PLDI)*, pages 203–213, 2001.

[2] A. Bertolino and M. Marré. Automatic generation of path covers based on the control flow analysis of computer programs. *IEEE Trans. Softw. Eng.*, SE-20(12):885–899, 1994.

[3] R. Boyer, B. Elspas, and K. Levitt. SELECT – a formal system for testing and debugging programs by symbolic execution. In *Proc. of the Int'l conf. on Reliable Software*, pages 234–245, 1975.

[4] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software – Practice And Experience*, 30:775–802, 2000.

[5] U. Buy, A. Orso, and M. Pezzè. Automated testing of classes. In *Proc. Int'l Symp. on Software Testing and Analysis (ISSTA)*, pages 39–48, 2000.

[6] W. Chan et al. Model checking large software specifications. *IEEE Trans. Softw. Eng.*, SE-24(7):498–520, 1998.

[7] T. Chen, T. Tse, and Z. Zhou. Semi-proving: an integrated method based on global symbolic evaluation and metamorphic testing. In *Proc. Int'l Symp. on Software Testing and Analysis (ISSTA)*, pages 191–195, 2002.

[8] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, SE-2(3):215–222, 1976.

[9] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzè. Using symbolic execution for verifying safety-critical systems. In *Proc. of 8th European Software Engineering Conf. and 9th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (ESEC/FSE)*, pages 142–151, 2001.

[10] P. Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2):324–328, 1996.

[11] P. Coward. Symbolic execution and testing. *Information and Software Technology*, 33:53–64, 1991.

[12] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proc. of the ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation (PLDI)*, pages 57–68, 2002.

[13] R. DeMillo and A. Offutt. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, SE-17:900–910, 1991.

[14] L. K. Dillon. Using symbolic execution for verification of Ada tasking programs. *ACM Transactions on Programming Languages and Systems*, 12(4):643–669, 1990.

[15] C. Flanagan et al. Extended static checking for Java. In *Proc. of the ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation (PLDI)*, pages 234–245, 2002.

[16] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *Proc. Int'l Symp. on Software Testing and Analysis (ISSTA)*, pages 53–62, 1998.

[17] N. Gupta, A. P. Mathur, and M. L. Soffa. Automated test data generation using an iterative relaxation method. In *Proc. of Int'l Symp. on Foundations of Software Engineering (FSE)*, pages 231–244, 1998.

[18] T. Hogg, B. Huberman, and C. Williams. Phase transitions and the search problem. *Artif. Intel.*, 81:1–15, 1996.

[19] R. Kemmerer and S. Eckmann. UNISEX: a UNIx-based Symbolic EXecutor for Pascal. *Software – Practice and Experience*, 15(5):439–458, 1985.

[20] J. King. Symbolic execution and testing. *Comm. of the ACM*, 19(7):385–394, 1976.

[21] B. Korel and A. Al-Yami. Assertion-oriented automated test data generation. In *Proc. 18th Int'l Conf. on Software Engineering (ICSE-18)*, pages 71–80, 1996.

[22] A. Mackworth. Constraint satisfaction. In *Encyclopedia of Artificial Intelligence*, volume 1, pages 205–211. John Wiley, New York, 1990.

[23] C. Michael, G. McGraw, and M. Schatz. Generating software test data by evolution. *IEEE Trans. Softw. Eng.*, SE-27(12):1085–1110, 2001.

[24] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Comm. of the ACM*, 33(12):32–44, 1990.

[25] A. Offutt, Z. Jin, and J. Pan. The dynamic domain reduction procedure for test data generation. *Software – Practice and Experience*, 29(2):167–193, 1999.

[26] R. Pargas, M. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9:263–282, 1999.

[27] N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. In *Proc. Int'l Symp. on Software Testing and Analysis (ISSTA)*, pages 73–81, 1998.

[28] M. R. Woodward, D. Hedley, and M. A. Hennell. Experience with path analysis and testing of programs. *IEEE Trans. Softw. Eng.*, SE-6(3):278–286, 1980.

[29] C. Xu and J. Zhang. EFSM generation for C programs with functions. In *Proc. Int'l Conf. for Young Computer Scientists (ICYCS)*, pages 90–94, 2001.

[30] D. Yates and N. Malevris. Reducing the effects of infeasible paths in branch testing. *ACM SIGSOFT Soft. Eng. Notes*, 14(8):48–54, 1989.

[31] J. Zhang. Symbolic execution of program paths involving pointer and structure variables. In *Proc. 4th Int'l Conf. on Quality Software (QSIC)*, 2004.

[32] J. Zhang and X. Wang. A constraint solver and its application to path feasibility analysis. *Int'l J. of Software Engineering and Knowledge Engineering*, 11(2):139–156, 2001.

[33] J. Zhang and C. Xu. Sequential program checking through constraint solving and symbolic execution of extended finite state machines. Technical Report ISCAS-LCS-02-14, Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Dec. 2002.