Symbolic Execution of Program Paths Involving Pointer and Structure Variables[*]

Jian Zhang
Laboratory of Computer Science
Institute of Software
Chinese Academy of Sciences
Beijing 100080, China
Email: zj@ios.ac.cn

Abstract

Many white-box testing methods are based on the analysis of program paths. For these methods, an important problem is to determine the feasibility of a given path, and find appropriate input data to execute the path if it is feasible. In this paper, the symbolic execution of program paths is studied. An approach is presented, which translates a path involving pointer and structure variables to a path involving simple variables and array variables only. The later is then analyzed with constraint solving techniques. An implementation of the translation tool is described with some examples. Preliminary experimental results show that the approach is quite efficient and applicable to paths in typical programs.

1. Introduction

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, a serious problem encountered is that a large portion of the paths can not be executed [9]. It is very important to identify such paths as early as possible. However, the path executability (feasibility) problem is undecidable in general. Thus we either use approximate analysis methods or restrict the syntax of program paths.

Previously we described a method and an automatic tool for analyzing paths which involve simple variables and array variables only [16]. The tool can decide the feasibility of paths in many benchmark programs. However, the inability to handle pointers is a severe limitation.

* This work was supported by the National Science Fund for Distinguished Young Scholars (No. 60125207).

This paper extends our early work such that pointer and structure variables can be handled. The paper is organized as follows. In the next section, we briefly recall some basic concepts and notations. Then we describe the main techniques for deciding the feasibility of program paths. In Section 4, we present our method for handling pointers and describe an automatic tool, together with some examples. Finally we compare the approach with other related methods and outline some future research directions.

2. Basic Concepts and Notations

A program can be described by a flow graph which is a directed graph, and a path can be defined as in graph theory. In this paper, we focus on unit testing. In other words, we assume that the program has only one function.

As in [16], we assume that a path can be represented by a sequence of assignment statements and conditional expressions (i.e., assertions), preceded by a suitable set of variable declarations. We also use a C-like syntax for presenting example programs or paths. Thus for example, the symbol '=' denotes assignment, while '==' denotes equality. An N-element array a consists of the elements a[0], a[1], ..., a[N-1].

A path is *executable* (or *feasible*) if there are input data such that the program is executed along that path. Otherwise, it is *unexecutable* (or *infeasible*). Let us look at a simple example.

Example 1. The following path is infeasible.

```
  int i, j;
@ (i > 5);
  j = 4;
@ (i + j < 8);
```

Here '@' denotes a conditional expression.

3. Deciding Path Feasibility by Symbolic Execution and Constraint Solving

In [16], we present an approach for deciding the feasibility of program paths, based on symbolic execution and constraint solving techniques. An automatic tool called PAT is described. Its input is a path extracted from a program, such as the one in the previous section. The variables can be integer or real variables, or arrays of them. Logical operators may occur in the conditional expressions, but nonlinear arithmetic expressions are not allowed. If the path is feasible, PAT will give initial values for the input variables such that the path can be executed. Otherwise, the answer will be "infeasible".

Essentially PAT first generates a *path condition* (PC) from the path, and then decides the satisfiability of the PC. The PC is a set of constraints on the input variables, such that the path is feasible if and only if the PC is satisfiable. It can be obtained by traversing the path either forwardly (i.e., from the entry node of the path to the exit node), or backwardly. The latter is called backward substitution, and is described in [16]. It is similar to the derivation of Dijkstra's weakest preconditions [5].

Alternatively, we can obtain the path condition by *symbolic execution* [2, 4, 12]. Its basic idea is very simple, namely, to "execute" a program using symbolic values for variables. Thus, for example, after the assignment $z = x + 2y$ is executed, the variable z will get a symbolic value like $a_0 + 2b_0$, where a_0 and b_0 are the initial values of x and y. During symbolic execution, the "values" of the variables are typically complicated expressions.

The symbolic execution algorithm works like this: we examine each statement or conditional expression in the path, from the beginning to the end. Initially, PC is empty. When we come across a conditional expression, we add it to PC, with each variable replaced by its "value". When we come across an assignment statement, we update the value of the variable on the left-hand side of the assignment. To decide path feasibility, we need to determine whether certain relationships can hold between the variables.

Let us consider Example 1 again. After the first step of execution, PC = { `i > 5` }. The next assignment gives the value 4 to `j`. Then the last conditional expression becomes `i + 4 < 8`. So the PC corresponding to the whole path is

{ `i > 5;  i + 4 < 8` }.

It is clearly unsatisfiable. Thus the path is infeasible.

In general, the PC may be more complicated than linear inequalities like the above. The variables can be of different types, and the constraints may have logical connectives like AND, OR, NOT. To solve the constraints, we combine (integer) linear programming with satisfiability checking for propositional logical formulas. Simply stated, our constraint can be regarded as a Boolean formula, but each Boolean variable may denote a primitive constraint over integer and real variables. Here a primitive constraint stands for a comparison between linear polynomials (e.g., `x + 2y > 3`, `2x - y = 5`). We implemented a backtracking algorithm to find the solutions to the Boolean formula. For each (partial) solution, we use a mixed integer linear programming package called `lp_solve` to check whether the primitive constraints hold. For more details about the constraint solving method, see [16].

Remark 1. In addition to testing, PAT can also be used for (partial) verification. We may attach the negation of the correctness property to the end of the path, as a "postcondition", and check whether the extended path is feasible. If yes, the program is wrong and we get a counterexample. Note however, that the postcondition can only be a conjunction of assertions.

Remark 2. PAT relies on `lp_solve` to deal with linear numerical constraints. Occasionally we get inaccurate results, because there may be rounding errors in the numerical computation.

4. Translating Program Paths Having Pointer and Structure Variables

Pointers cause difficulties to programming and program analysis. Most symbolic execution systems do not accept programs having pointer variables. We deal with this problem by translating a program path P_0 involving pointers and structures into a path P_{new} which has simple variables and array variables only. The general idea is to model the memory by a set of auxiliary variables, which may be regarded as forming a virtual array `av[]`.

For simplicity, we assume that the simple variables are all integer variables. We do not consider floating-point numbers, characters or the enumeration type in this paper. Thus one unit of memory corresponds to 4 bytes, since in most cases, `sizeof(int) = sizeof(p) = 4`. Here `p` is a pointer variable.

4.1. The Input Path

As previously, we use a C-like language to describe programs. But in a C program, the expressions can be very complicated. In this paper, we assume that the pointer-related language constructs are very simple ones, like `&x`, `*p`, `p->f` and so on. Note that `p->next->f` is not allowed. This limitation may be remedied by introducing new pointer variables. The `malloc()` function is allowed, but `free()` is not.

The Appendix gives the syntax of the input path in the extended BNF notation.

4.2. Details of Translation

Now we describe our translation method. Initially, each variable is assigned an index in the array av, according to its order of declaration. The smallest index is 1 rather than 0, because 0 is reserved for the constant `NULL`. Let us denote the index of variable `x` by `Idx(x)`. For a simple example, suppose we have the following variables in the program:

```
int  i, j, *p;
```

Then `Idx(i)` = 1, `Idx(j)` = 2 and `Idx(p)` = 3. In other words, the variables `i`, `j` and `p` will be replaced by the auxiliary variables `av[1]`, `av[2]` and `av[3]`, respectively.

The elements of the array `av[]` are not necessarily adjacent physically. One exception is that when a block of memory is allocated dynamically, the corresponding auxiliary variables are neighbors of each other. When we come across the statement `p = malloc(n)`, we introduce a set of `n` new auxiliary variables. The index of the first variable is assigned to `p`, namely `av[Idx(p)]`.

In general, expressions in the path are translated in the following way:

- The simple variable `x` is mapped to `av[Idx(x)]`.
- The expression `&x` is mapped to `Idx(x)`.
- The expression `*p` is mapped to `av[av[Idx(p)]]`.
- The expression `s.Fk`, where `s` is a structure variable containing the fields `Fi` ($0 \leq i < n$), is mapped to `av[Idx(s)+k]`.
- The expression `p->Fk`, where `p` is a pointer variable pointing to a structure containing the fields `Fi` ($0 \leq i < n$), is mapped to `av[av[Idx(p)]+k]`.

Let us look at the following simple example:

```
 int n, *p;

 p = & n;
*p = 5;
```

After the variable declarations, `Idx(n)` = 1, `Idx(p)` = 2. So the new path (i.e., the result of translation) is the following:

```
av[2] = 1;   av[av[2]] = 5;
```

The first assignment statement sets `p`'s value to `Idx(n)` = 1. After the two assignment statements, the variable `n` (i.e., `av[1]`) gets the value 5. We can see that the aliasing problem is solved by making the "addresses" of the variables explicit.

4.3. An Automatic Tool

Based on the above ideas, we have implemented a translation tool in C. Its input is a program path which may involve pointer and structure variables, and its output is a path involving simple variables and array variables only. The latter can be analyzed using PAT.

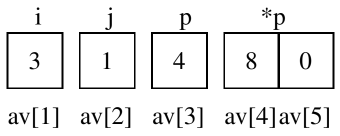
Example 2. Suppose we are given the following path:

```
typedef struct mySt *pms;
struct mySt {
  int val;  pms next;
};

int i, *j;
pms p;

{
  j = & i;
 *j = 3;
  p = malloc(2);
  p->val = 8;
  p->next = NULL;
}
```

The result of translation is the following sequence:

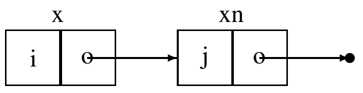
```
av[2] = 1;
av[av[2]] = 3;
av[3] = 4;
av[av[3]] = 8;
av[av[3]+1] = 0;
```

After executing the new path, the memory store is like this:



Simple as our language is, it is enough to represent many algorithms, including the list processing routines in [13].

Example 3. Let us look at a sorting routine in [13], i.e., `insert_sort_b2()`, which is a wrong version of the insertion sort. To check its correctness, we may examine a number of its paths. For each path, we assume that it operates on a list of two integers, like the following:



We attach to the end of the path with a postcondition saying that the first integer is greater than the second integer. If the path is feasible, we will get a counterexample because the postcondition violates the correctness property. The following is one of the extended paths.

```
typedef struct node *List;
struct node {
  int  data;
  List next;
};

int  i, j;
List x, xn;
List r, pr, rn, l, pl;

{
  // the initial list
  xn = malloc(2);
  xn->next = NULL;
  scanf("%d",&j);
  xn->data = j;
  x = malloc(2);
  x->next = xn;
  scanf("%d",&i);
  x->data = i;

  // path
@ x != NULL;
  pr = x;
  r = x->next;
@ r != NULL;
  pl = x;
  rn = r->next;
  l = x->next;
@ (l == r);
  pr = r;
  r = rn;
@ (r == NULL);

  // postcondition
@ (x->data > xn->data);
}
```

Given the above as input, our tools find the following data within a fraction of a second: INPUT1 = 0, INPUT2 = 1. It means, if the first input value is 0 (which is assigned to the variable j), and the second input value is 1 (which is assigned to the variable i), the postcondition will be satisfied and the path is executable. In other words, the list { 1, 0 } is a counterexample to the sorting routine.

We have tried our tools on other similar program paths which have pointer variables. The results are encouraging. In most cases, the running time is typically a few milliseconds on a state-of-the-art computer.

One feature of our approach is that it can deal with paths involving pointer and integer variables, which have both numerical and logical operations.

Example 4. Our tools can determine that the following path is infeasible.

```
int i, j;
int *p;
{
  scanf("%d",&i);
  scanf("%d",&j);
@ (i > 2) && (j > 2);
  p = & i;
  *p = *p + j;
@ *p < 2;
}
```

4.4. Discussion

Although our tools can be used to analyze a variety of program paths, as demonstrated by the previous examples, we should mention some limitations here.

Firstly, the `free` statement is not allowed in the input language. The main reason is to reduce the complexity of the analysis. We are mainly concerned with the functional correctness of programs, rather than memory leak and similar problems. For the latter, there are already quite a few good tools such as PREfix [3].

Secondly, all the auxiliary variables are of the same type. We expect that the compiler or other analysis tools can detect type errors like the addition of two pointers (e.g., `p+q`). If the input path involves other data types, the translation is more complicated. We need to introduce more than one auxiliary arrays.

The last problem is a side-effect of the assumption that the `av` variables are adjacent. Suppose there are two statements:

```
p = malloc(5);   q = malloc(4);
```

If the user performs `p++` for 8 times, it would be an error. However, under our translation, this is valid. Again, it is expected that the user applies other tools to find this kind of errors, before using our method.

In addition to the above, we assume that the `malloc` function never fails. We do not allow explicit allocation of variable-sized blocks, either.

5. Related Work

As we mentioned in the Introduction, determining path feasibility is a key problem in path-oriented program analysis and test data generation. Many previous works either avoid this problem or use approximate methods. For example, in [15], 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.

More recently, some authors have proposed various accurate analysis methods. For instance, a powerful method for deciding the feasibility of paths is described in [8]. It is especially suitable for numerical computation applications. But logical expressions seem to be difficult for the method.

A path exploration tool called PET [7] uses a decision procedure for the Presburger arithmetic to decide path feasibility, but it is well-known that such a decision procedure suffers from high complexity. That is one reason why we restrict the postconditions to simple assertions (rather than first-order formulas with quantifiers).

Most previous symbolic execution systems also have difficulty with logical expressions. For example, an important module of the UNISEX system [11] is the theorem prover, which was “a stub that asks the user to make the reduction” (page 447 of [11]). 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 [11]).

A test data generation technique called dynamic domain reduction (DDR) is proposed by Offutt *et al.* [14]. But their tool “has only been applied to numeric software”, because it “does not handle pointers and the expression handling is limited to expressions that use numeric operators” (page 187 of [14]).

In the programming language and software engineering community, much attention has been paid to the analysis of programs having pointers. But most of these methods are concerned with the points-to relationships between variables and other potential memory problems. They are aimed at analyzing large programs and do not consider the complete bahavior of programs.

To quote Lev-Ami *et al.* (page 28 of [13]), their approach “intentionally ignores the specific values of the `d`- and the `n`-components (an int and a memory address, respectively), and just records certain relationships that hold among the elements”. Hind and Pioli [10] performed an empirical study of five pointer analysis algorithms on C programs. The worst-case complexity of each algorithm ranges from linear to polynomial. In contrast, the worst-case complexity of our analysis algorithm is exponential. However, our approach records and analyzes the (symbolic) values of all variables, and it can get more accurate results.

Elgaard *et al.* [6] propose to verify a program by transforming it into a formula in second-order logic, and then using the `MONA` tool to decide the formula’s validity. The program may have pointer variables and assertions (preconditions and postconditions). However, integer arithmetic is not supported.

Our techniques are much simpler, if compared with those used in compilers [1], because our input language is simpler, as indicated in Section 4.1 and Section 4.4. We think that there is some tradeoff between the efficiency of the analysis and the expressiveness of the language. A compiler typically keeps too much information, which makes it difficult to perform an accurate analysis.

6. Concluding Remarks

Path feasibility is very important to white-box testing. It can also be used for verifying the correctness of a program (if the program has only a finite number of paths, and the negation of the correctness property can be expressed using simple assertions).

In this paper, we have extended our previous work on path analysis, such that pointer and structure variables can be handled. Our aim is to analyze *small* programs efficiently and accurately. An automatic tool is implemented, which translates the input path to a format which is acceptable to PAT [16]. To our knowledge, no other tools can decide the feasibility of paths like that of Example 4 automatically. The current version of the translation tool is just a research prototype, which demonstrates the applicability of our approach. But it seems to be quite efficient.

However, the input language is very simple. In the future, we shall try to include more data types (such as floating pointer numbers) and more complex expressions (such as arrays of pointers and pointers to arrays). Our method also has certain limitations, as mentioned in Section 4.4. However, some of them are not fundamental. For example, we could have introduced several arrays of auxiliary variables, one for pointers to integers, another for pointers to floating-point numbers, and so on.

Acknowledgements

The author is indebted to the anonymous reviewers for their comments and suggestions, and to Guangyuan Li for helpful discussions.

References

[1] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques and Tools,* Addison-Wesley, 1986.

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

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

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

[5] E.W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976.

[6] J. Elgaard, A. Møller and M.I. Schwartzbach, Compile-time debugging of C programs working on trees, *Proc. European Symp. on Programming (ESOP)*, LNCS 1782, 119–134, 2000.

[7] E.L. Gunter and D. Peled, Path exploration tool, *Proc. of the 5th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems* (*TACAS'99*), LNCS 1579, 405–419, 1999.

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

[9] D. Hedley and M. A. Hennell, The causes and effects of infeasible paths in computer programs, *Proc. 8th Int. Conf. on Software Engineering*, 259–266, 1985.

[10] M. Hind and A. Pioli, Which pointer analysis should I use?, *Proc. Int. Symp. on Software Testing and Analysis (ISSTA)*, 113–123, 2000.

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

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

[13] T. Lev-Ami, T. Reps, M. Sagiv and R. Wilhelm, Putting static analysis to work for verification: A case study, *Proc. Int. Symp. on Software Testing and Analysis (ISSTA)*, 26–38, 2000.

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

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

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

Appendix. Syntax of the Path

The following is the syntax for the input path. Here { `x` } means zero or more repetitions of `x`; and | means a choice among alternatives. We use quotes around the tokens to denote strings or characters (terminal symbols), so as to distinguish them from the meta-symbols.

$\langle inpath\rangle$ ::= $\langle typs\rangle$ $\langle vars\rangle$ '`{`' $\langle stmtL\rangle$ '`}`'
$\langle typs\rangle$ ::= { $\langle typ1\rangle$ }
$\langle typ1\rangle$ ::= "`typedef`" $\langle typ0\rangle$ $\langle stars\rangle$ IDEN '`;`'
  | "`struct`" IDEN '`{`' $\langle vars\rangle$ '`}`' '`;`'
$\langle typ0\rangle$ ::= "`int`"
  | "`struct`" IDEN
$\langle vars\rangle$ ::= { $\langle var1\rangle$ }
$\langle var1\rangle$ ::= $\langle typid\rangle$ $\langle idens\rangle$ '`;`'
$\langle typid\rangle$ ::= "`int`"
  | IDEN
$\langle idens\rangle$ ::= $\langle iden1\rangle$ { '`,`' $\langle iden1\rangle$ }
$\langle iden1\rangle$ ::= $\langle stars\rangle$ IDEN
$\langle stars\rangle$ ::= { '`*`' }
$\langle stmtL\rangle$ ::= { $\langle stmt1\rangle$ }
$\langle stmt1\rangle$ ::= $\langle scanf\rangle$
  | $\langle asgmt\rangle$
  | $\langle asrtn\rangle$
$\langle asrtn\rangle$ ::= '`@`' $\langle expre\rangle$ '`;`'
$\langle asgmt\rangle$ ::= $\langle vrb\rangle$ '`=`' $\langle rhsa\rangle$ '`;`'
$\langle rhsa\rangle$ ::= '`&`' IDEN
  | "`malloc`" '`(`' NUM '`)`'
  | $\langle smpex\rangle$
$\langle expre\rangle$ ::= $\langle smpex\rangle$ $\langle relop\rangle$ $\langle smpex\rangle$
  | $\langle smpex\rangle$
$\langle smpex\rangle$ ::= $\langle term\rangle$ $\langle tmop\rangle$ $\langle term\rangle$
  | $\langle term\rangle$
$\langle tmop\rangle$ ::= '`+`' | '`-`' | "`||`"
$\langle term\rangle$ ::= $\langle factr\rangle$ "`&&`' $\langle factr\rangle$
  | $\langle factr\rangle$
$\langle factr\rangle$ ::= $\langle vrb\rangle$ | $\langle cplx\rangle$
$\langle cplx\rangle$ ::= '`(`' $\langle expre\rangle$ '`)`'
  | '`!`' $\langle factr\rangle$
  | "`NULL`"
  | NUM
$\langle vrb\rangle$ ::= '`*`' IDEN
  | IDEN "`->`" IDEN
  | IDEN "`.`" IDEN
  | IDEN

There are several undefined symbols:

- `IDEN` denotes an identifier;
- `NUM` denotes a number;
- $\langle scanf\rangle$ denotes an input statement; and
- $\langle relop\rangle$ denotes a relational operator (such as greater-than, equal-to).