Available online at www.sciencedirect.com
ScienceDirect
Information Processing Letters 107 (2008) 87–92
Information Processing Letters
www.elsevier.com/locate/ipl

An efficient method to generate feasible paths for basis path testing ☆

Jun Yan, Jian Zhang *

State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, PR China

Received 18 May 2007; received in revised form 14 January 2008; accepted 14 January 2008

Available online 13 February 2008

Communicated by J.L. Fiadeiro

Abstract

Basis path testing is a very powerful structural testing criterion. The number of test paths equals to the cyclomatic complexity of program defined by McCabe. Traditional test generation methods select the paths either without consideration of the constraints of variables or interactively. In this note, an efficient method is presented to generate a set of feasible basis paths. The experiments show that this method can generate feasible basis paths for real-world C programs automatically in acceptable time.

Keywords: Software engineering; Basis path testing; Path feasibility; Test data generation

1. Introduction

Structural testing (white box testing) is a widely used class of program testing methods. Many test criteria, such as statement coverage and branch coverage, need to find a set of executable paths which meet some requirements. To execute each path, we need to find appropriate values for the input variables. This problem is called test data generation. If a path cannot be exercised by any set of input data, we say the path is *infeasible*. Hedley and Hennell's empirical work [1] indicated that a great majority of program paths are found to be infeasible for some library programs. They also pointed out that some algorithms for generating test paths without considering the path feasibility "seem to be of little value".

Basis path testing [2] is a famous structural testing criterion. This criterion requires to cover a basis set of paths, where any path of the program under test is a linear combination of these basis paths. The number of basis paths equals to the cyclomatic complexity [2] of the program. This criterion subsumes the branch coverage and statement coverage [3]. Since we need only a few number (which equals to the cyclomatic complexity) of paths to satisfy this coverage criterion, this test methodology is very important in program testing. However, the existing test generation methods for this criterion are either interactive or they have not taken into account the path feasibility. Therefore the previous works of basis path testing seem to be hard to be applied to generate basis path test suites automatically.

☆ This work is partially supported by the National Natural Science Foundation of China (NSFC) under grant number 60633010 and the National High-Tech Research (863) program under grant number 2006AA01Z402.

* Corresponding author. Address: 4# South Fourth Street, Zhong Guan Cun, P.O. Box 8718, Beijing 100190, PR China.

E-mail addresses: yanjun@ios.ac.cn (J. Yan), zj@ios.ac.cn (J. Zhang).

URLs: http://lcs.ios.ac.cn/~yanjun (J. Yan), http://lcs.ios.ac.cn/~zj (J. Zhang).

0020-0190/$ – see front matter
doi:10.1016/j.ipl.2008.01.007

This paper presents an efficient method to generate a set of feasible basis paths. We try to apply this method to some real-world C functions, and the experiments show that this method can generate feasible basis paths automatically in acceptable time.

2. Background

We will give some backgrounds in this section, mainly about some notations and the basis path testing.

2.1. Structure and path of program

Many forms of structural testing make reference to the Control Flow Graph (CFG) of the program in question. A CFG is a directed graph that consists of a set $\mathcal{N}$ of nodes and a set $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ of directed edges between nodes. Each node represents a linear sequence of computations. Each edge $e = \langle n_i, n_j \rangle \in \mathcal{E}$ is a transfer of control from node n_i to node n_j, and is associated with a predicate that represents the condition of control transfer from n_i to n_j. In a CFG, there is a unique entry node $s \in \mathcal{N}$ and a unique exit node $f \in \mathcal{N}$. A path through a control flow graph is a sequence of nodes $A = n_{a_1} n_{a_2} \cdots n_{a_{m+1}}$ where $n_{a_1} = s$ and $n_{a_{m+1}} = f$, or alternatively a sequence of edges $A = e_{a_1} e_{a_2} \cdots e_{a_m}$, where m is the *length* of path A and $e_{a_i} = \langle n_{a_i}, n_{a_{i+1}} \rangle$ $(0 < i \leqslant m)$. The set of all the paths of a program is denoted as $\mathcal{P}$.

Example 1. Consider the following C function which has two decisions, "if" and "while".

```
int y;
void foo(int x){
/*S1*/ if (x < 5)
/*S2*/   y = 2;
/*S3*/ while (x < 5)
/*S4*/   x++;
/*S5*/ return;
}
```

The control flow graph can be seen in Fig. 1. We use the expressions after "@" to represent the predicates of edges. Obviously the path $e_1 e_2 e_6$ is infeasible.

The path feasibility problem is undecidable in general. But if we restrict the constraint of the path (for example, only boolean and linear arithmetic constraints), the problem becomes solvable. Zhang and Wang [4] used a symbolic execution method [5] to decide the feasibility of a given path of program. This method first collects the constraints from a path by backward substitution. The path is feasible if and only if its constraints are satisfiable. Then a constraint solver called BoNuS is employed to decide the satisfiability of the constraints.

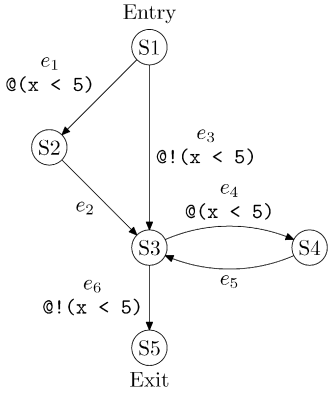

Fig. 1. Control flow graph.

Based on this work, we developed a prototype tool called SimC [6]. It first parses the C code into an intermediate representation and builds up CFG which has unique entry and exit nodes. Then it uses a BFS (Breadth-First Search) algorithm to traverse the CFG from the entry node and extracts all the test paths no longer than a given length limit from CFG. At last SimC calls the constraint solver to decide the feasibility of the paths. This tool supports most C features, including basic control logics such as `if-else` decision and `while` loop, and some complex data structures such as pointers, array and structure operations. It uses some conditional expressions (i.e., stubs) to simulate the behavior of the called function. In this paper, we employ SimC to construct the CFG for C functions and extract paths from the CFGs.

2.2. Basis path testing

Let $\mathcal{E} = \{e_1, e_2, \ldots, e_{|\mathcal{E}|}\}$, where $|\mathcal{E}|$ is the size of set $\mathcal{E}$. The basis path coverage criterion only considers the coverage of edges. Then each path A of CFG G can be represented as a vector $A = \langle a_1, a_2, \ldots, a_{|\mathcal{E}|} \rangle$, where a_i $(1 \leqslant i \leqslant |\mathcal{E}|)$ is the number of times that e_i appears in path A. We can define operations on the paths. Let $A = \langle a_1, a_2, \ldots, a_{|\mathcal{E}|} \rangle$ and $B = \langle b_1, b_2, \ldots, b_{|\mathcal{E}|} \rangle$, then

$$A + B = \langle a_1 + b_1, a_2 + b_2, \ldots, a_{|\mathcal{E}|} + b_{|\mathcal{E}|} \rangle,$$

$$\lambda A = \langle \lambda a_1, \lambda a_2, \ldots, \lambda a_{|\mathcal{E}|} \rangle.$$

A path B is said to be the *linear combination* of path set $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ (or we say B can be *linearly*

represented by $\mathcal{A}$) if and only if there exist coefficients $\lambda_1, \lambda_2, \ldots, \lambda_n$ such that

$$B = \lambda_1 A_1 + \lambda_2 A_2 + \cdots + \lambda_n A_n.$$

A set of paths is said to be *linearly independent* if each path in it cannot be linearly represented by other paths in the set. Based on these two concepts, we can give the definition of basis path set, which is the test set of basis path testing:

Definition 2 *(Basis path set).* A linearly independent path set $\mathcal{B}$ of a program is said to be a basis path set of the program if any path P of the program's CFG can be linearly represented by $\mathcal{B}$.

Basis path testing is first introduced by McCabe [2]. He also proved that the size of basis path set is unique for any given graph and is called the cyclomatic complexity $v(G)$ of the program. The cyclomatic complexity can be easily calculated by the following formula:

$$v(G) = e - n + 2 = d + 1,$$

where e, n, d are the number of edges, vertices and condition nodes (all of which are binary decisions), respectively. The equation indicates that this complexity depends only on the decision structure of a program.

We can easily deduce that any basis path set $\mathcal{B}$ is the maximal linearly independent subset of $\mathcal{P}$, otherwise there exists at least one path $p \in \mathcal{P} \setminus \mathcal{B}$ such that p is linearly independent of the paths of $\mathcal{B}$, which contradicts the definition of basis path set. Therefore $r(\mathcal{P})$ (the rank of $\mathcal{P}$) equals to $v(G)$.

Example 3. Consider the C function `foo` of Example 1. There are six edges in the CFG, and the program's cyclomatic complexity is 3. Without considering the feasibility of the paths, we find the following three paths (generated by Poole's method, see Section 6) which are linearly independent:

$$P_1 = e_3 e_6 = \langle 0, 0, 1, 0, 0, 1 \rangle,$$
$$P_2 = e_1 e_2 e_6 = \langle 1, 1, 0, 0, 0, 1 \rangle,$$
$$P_3 = e_3 e_4 e_5 e_6 = \langle 0, 0, 1, 1, 1, 1 \rangle.$$

Any path of the CFG can be linearly represented by these three paths. For instance, the path

$$P = e_1 e_2 e_4 e_5 e_4 e_5 e_6 = \langle 1, 1, 0, 2, 2, 1 \rangle$$

can be represented as $P = -2P_1 + P_2 + 2P_3$.

However, the paths P_2 and P_3 are both infeasible and cannot be used as test cases. We need a method to find feasible basis paths.

```
BOOL BPGen(int UL){
  B = φ;
  for(len = 1; len ⩽ UL; len++)
    while((P = select(len))! = NULL)
      if (B == φ || !LR(P, B))
        if (P is feasible) {
          add P to B;
          if (|B| == v(G)) return TRUE;
        }
  return FALSE;
}
```

Fig. 2. Basis path generation algorithm.

3. The proposed basis path generation method

Since we can decide the feasibility of a given path, naturally, generating feasible test paths can be divided into two steps:

(i) Generate a finite set of feasible paths $\mathcal{F}$ which satisfies the basis path coverage criterion.
(ii) Find a minimal subset $\mathcal{S}$ of set $\mathcal{F}$ such that $\mathcal{S}$ satisfies the test coverage. Then $\mathcal{S}$ is the test path set.

However, these two steps are not efficient. The first step should check the feasibility of all the paths and the feasibility checking is quite time-consuming compared with other operations. In fact we can select basis paths on-the-fly. A path belonging to the basis path set should satisfy two conditions:

(i) It should be feasible, and
(ii) It should be linearly independent of all other selected paths.

So we propose the algorithm in Fig. 2 to get the basis path set. The upper bound of path lengths is set as *UL*, which is specified by the user (refer to Section 4 for why we should set this bound). The subroutine select(int *len*) employs SimC to select the next path with length *len*, and if it cannot find a proper path, it returns NULL. And the function LR(P, $\mathcal{B}$) returns TRUE if the path P can be linearly represented by path set $\mathcal{B}$. This can be checked by some linear programming tool such as `lp_solve` [7]. The main idea of the algorithm is to select the maximal linearly independent subset of $\mathcal{P}$.

The correctness of this algorithm can be easily proved. The feasible path set $\mathcal{B}$ is the maximal linear independent subset of all the paths generated (denoted as $\mathcal{F}$). If the algorithm BPGen returns TRUE (i.e., $r(\mathcal{F}) = v(G)$), $\mathcal{B}$ will be the maximal linear independent subset of $\mathcal{P}$, thus is the basis path set. The condition of $r(\mathcal{F}) < v(G)$ will be discussed in Section 4.

Table 1
Program paths

Path ID	Path sequence	Length	Test data
1	e_3e_6	2	x = 5
2	$e_1e_2e_6$	3	infeasible
3	$e_3e_4e_5e_6$	4	infeasible
4	$e_1e_2e_4e_5e_6$	5	x = 4
5	$e_3e_4e_5e_4e_5e_6$	6	infeasible
6	$e_1e_2e_4e_5e_4e_5e_6$	7	x = 3

The efficiency of this algorithm mainly depends on the time of calling feasibility checker. If all the paths are feasible, we only need to call the constraint solver $v(G)$ times. If some of the paths are infeasible, the number of calling times will increase. Since $v(G)$ is a small number (McCabe suggested a reasonable upper bound of $v(G)$ is 10 for program units [2]), the performance of this algorithm is acceptable for unit testing (see Section 5).

Example 4. We can apply our method to the program of Example 1. For this simple CFG, we set $UL = 20$. Our algorithm enumerates six paths (listed in Table 1), three of which are feasible, and returns TRUE. Since $v(G) = 3$, the paths 1, 4 and 6 form the basis path set.

4. Discussion

The data dependencies between the control structures can prevent some paths of the CFG from being exercised. This is the cause of infeasible paths. Sometimes these data dependencies do not affect the cyclomatic complexity. For instance, we can still find three feasible basis paths for Example 1 in which the two decisions are strongly dependent. But in some other cases, we cannot always get $v(G)$ basis paths. Here is an example from [8]:

```
void f(int x){
  int y, z;
  if (x < 5) y = 2;
  if (x < 5) z = 1;
}
```

In fact we can only find two feasible paths for this program whose cyclomatic complexity is 3.

The actual complexity, denoted as *ac* of a test process is defined as the maximal number of linearly independent paths that have been executed during testing, or more formally as the rank of the set of paths that have been executed during testing.

The realizable complexity, denoted as *rc*, is the maximum possible actual complexity, i.e. the rank of the set of paths induced by all possible tests. For the previous example `f(x)`, we have $ac = rc = 2$.

There is an obvious relation: $ac \leqslant rc \leqslant v(G)$. For our method, if $ac < v(G)$, one of the two situations can hold:

(i) At least one additional independent test can be executed, i.e., we have given a too small *UL*.
(ii) $ac = rc$, and hence $rc < v(G)$.

Calculating rc is theoretically undecidable [9]. Thus if the function BPGen() returns FALSE, we do not know which situation holds. Since the number of paths may be infinite and we cannot enumerate all of them, the user should give an upper bound *UL* on the length of paths to avoid infinite execution of our algorithm. And under this circumstance our method will construct a test set with *ac* test paths.

5. Experiments

We developed a tool based on SimC to generate feasible basis paths for C functions automatically. All the programs under test have no "goto" statements in the program. SimC is used as the front end to parse ANSI C program and generate execution paths. SimC and our test path generator are integrated by some Perl programs. In this section, we use some examples to show the performance of our tool. All the instances are run on a Pentium IV PC with 3.2 GHz CPU and 1 GB memory.

5.1. An example

Example 5. The function `getop` was used by Bertolino and Marré [10] as an example to generate test paths from CFG. The CFG can be seen in the paper [10]. This program has complex control structures. We can see from the CFG that it has 10 decision nodes and thus $v(G) = 11$. Also the paper [10] and our previous work [6] indicated that many of the paths for this program are infeasible. We apply our tool to this instance and find 11 basis paths. Our previous work [6] demonstrated that we need 4 paths to cover all of the edges in the CFG. In general, since basis path criterion subsumes the branch coverage, we need more test cases.

5.2. Some benchmarks

In our previous work [6], we tried SimC to find feasible test suite to satisfy branch coverage (i.e. covering all the edges of CFG) for some programs. Some of these programs come from GNU `coreutils` package. We

Table 2
Some benchmarks

Function	$v(G)$	$\|\mathcal{B}\|$	Max. len.
`getop()`	11	11	16
`strol()`	7	7	7
`InsertionSort()`	5	5	24
`dosify()`	8	8	25
`bsd_split_3()`	6	6	41
`attach()`	5	5	6
`remove_suffix()`	4	3	30
`quote_for_env()`	4	4	6
`isint()`	9	9	39

now use our tool to find feasible basis path test set for these benchmark programs. We list the number and maximal length of feasible basis paths in Table 2. For all these instances, $UL = 100$ is enough. All benchmarks can be processed in 10 seconds.

For the instance `remove_suffix()`, we get $rc < v(G)$, so we try to increase UL. But our tool cannot get more than 3 basis paths not longer than 100. Then we try to examine the program. This program includes the following segment which contains data dependency:

```
char *np, *sp;
np = name + 9; sp = suffix + 9;
while (np > name && sp > suffix) {
  np--, sp--;
  ...
}
if (np > name) ...
```

In fact, the condition `if(np > name)` cannot be satisfied, i.e., the actions of this `if` statement will never be executed. We can remove the `if` statement and our tool reports the complexity of modified program is $v(G') = rc = 3$. From this analysis, we can conclude that for the program `remove_suffix()`, $v(G) = 4$, $rc = 3$, we can only generate 3 feasible basis paths.

The experimental value of UL depends on the programming style of the code. For the program under test in this section, $UL = 100$ is enough, therefore in our tool UL is set as 100 by default. If $rc < v(G)$ occurs, the tester can choose to increase UL, or refine the source code along the infeasible paths which are linearly independent of the generated test paths to eliminate data dependency to get shorter feasible test paths.

6. Related work

There are some existing works about basis path generation from CFG. Joseph Poole [8] proposed a simple, automatic method to find basis paths from CFG of a program. The main idea is that for all the edges outgoing a decision node, label one as "default edge"; the other edges will only be traversed once during the search. This method will exactly generate $v(G)$ test paths, but the paths may be infeasible. Consider the CFG of Example 1, if we mark e_3 and e_6 as default, we will get 3 basis paths: $P_1 = e_3e_6$, $P_2 = e_1e_2e_6$ and $P_3 = e_3e_4e_5e_6$, but the latter two paths are both infeasible. The paper [11] presented another graph based method to find basis paths, which also does not take the path feasibility into account.

The baseline method [9] first selects a basis set of edges, then uses that set to generate successive paths, and finally uses the resulting paths/edges restricted to basis columns to get basis paths. To guarantee the feasibility of these basis paths, the tester has to choose between various alternatives when generating paths. Therefore the baseline method in practice is a semi-automatic one. Watson [12] presented a dynamic method to generate basis paths. Similar to our technique, this method also generates a feasible candidate path set (called minimal sufficient set of tests). They use some dynamic testing method to record some runtime traces as candidate paths. Different from our static symbolic-execution method, the dynamic method cannot assure that it can get a sufficient candidate set (i.e., 100% basis path coverage).

The method proposed in this paper relies on the availability of a precise path feasibility analyser which uses symbolic execution and constraint solving techniques. It can be used to find bugs in programs or generate test data. It may also be combined with static analysis to sharpen the analysis results. However, few previous works addressed path selection issues like basis path testing. The PathCrawler tool [13] combines static and dynamic analysis for generating test cases to cover all the feasible execution paths. Different from SimC, PathCrawler does not construct CFG and enumerate all the paths, but use a constraint-based method to cover the whole input space of the program under test. PathCrawler can generate test cases satisfying the all-paths and loop count-k criteria [3]. The basis path selection algorithm introduced in this paper, with proper modifications and improvements, can also be used to generate basis paths based on PathCrawler.

7. Conclusion

We discussed the test generation method for basis path testing in this paper. The existing test data generation methods for this test criterion are either interactive or do not take the path feasibility into account.

We presented an automatic method to generate feasible basis paths. For the program whose realizable complexity $rc = v(G)$, our method can definitely get a set of feasible basis paths with proper path length restriction. In practice, the length restriction for unit testing is suggested to be not more than 100. For the program with $rc < v(G)$, our method can generate a set of feasible test paths, whose size is rc.

References

[1] D. Hedley, M.A. Hennell, The causes and effects of infeasible paths in computer programs, in: Proc. 8th Int. Conf. on Software Engineering (ICSE'85), 1985, pp. 259–266.

[2] T.J. McCabe, A complexity measure, IEEE Trans. Softw. Eng. SE-2 (4) (Dec. 1976) 308–320.

[3] H. Zhu, P.A.V. Hall, J.H.R. May, Software unit test coverage and adequacy, ACM Comput. Surv. 29 (4) (Dec. 1997) 366–427.

[4] J. Zhang, X. Wang, A constraint solver and its application to path feasibility analysis, Internat. J. Software Engrg. Knowledge Engrg. 11 (2) (2001) 139–156.

[5] J.C. King, Symbolic execution and program testing, Comm. ACM 19 (7) (1976) 385–394.

[6] Z. Xu, J. Zhang, A test data generation tool for unit testing of C programs, in: Sixth International Conference on Quality Software (QSIC'06), 2006, pp. 107–114.

[7] lp_solve, http://groups.yahoo.com/group/lp_solve/.

[8] J. Poole, A method to determine a basis set of paths to perform program testing (NISTIR 5737), Tech. rep., Department of Commerce, NIST, Nov. 1995.

[9] A.H. Watson, T.J. McCabe, Structured testing: A testing methodology using the cyclomatic complexity metric, Tech. rep., Computer Systems Laboratory, NIST, NIST Special Publication 500-235, Sep. 1996.

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

[11] Z. Guangmei, C. Rui, L. Xiaowei, H. Congying, The automatic generation of basis set of path for path testing, in: Proceedings of the 14th Asian Test Symposium (ATS'05), 2005, pp. 46–51.

[12] A.H. Watson, Structured testing: Analysis and extensions, PhD thesis, Department of Computer Science, Princeton University, Nov. 1996.

[13] N. Williams, B. Marre, P. Mouy, M. Roger, PathCrawler: Automatic generation of path tests by combining static and dynamic analysis, in: EDCC, 2005, pp. 281–292.