Abstract Path Testing with PathCrawler

Nicky Williams
CEA, LIST, Laboratoire Sûreté des Logiciels
F-91191 Gif-sur-Yvette
France
00 33 169089472
nicky.williams@cea.fr

ABSTRACT

PathCrawler is a tool developed by CEA List for the automatic generation of test inputs to ensure the coverage of all feasible execution paths of a C function. Due to its concolic approach and depth-first exhaustive search strategy implemented in Prolog, PathCrawler is particularly efficient in the generation of tests to cover the fully expanded tree of feasible paths. However, for many tested functions this coverage criterion demands too many tests and a weaker criterion must be used. In order to efficiently generate tests for a new criterion whilst still using a concolic approach, we must modify the search strategy. To facilitate the definition and comparison of different coverage criteria, we propose a new type of tree, trees of abstract paths, and define the different types of abstract node in these trees. We demonstrate how several criteria can be conveniently defined in terms of coverage of these new trees. Moreover, efficient generation of tests to satisfy these criteria using the concolic approach can be designed as different strategies to explore these trees.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging – *Testing tools (e.g., data generators, coverage testing).*

General Terms

Design, Performance, Algorithms.

Keywords

structural testing, test generation, coverage criteria.

1. INTRODUCTION

PathCrawler [1][2] was one of the first test input generators to use a combination of concrete data and symbolic execution. In the literature, similar test generation tools are variously referred to as concolic, dynamic-symbolic-execution (DSE) or constraint-based execution tools. Below we will call them *concolic* tools. Unlike some concolic tools, such as [3], PathCrawler does not use concrete values to generate over-approximate path predicates.

.

However, PathCrawler is concolic in that, like these tools and [4], [5], [11], PathCrawler recovers the trace of each generated test and uses it to generate a prefix of the path predicate of the next test. This is an efficient way to generate tests for all-feasible-path coverage, the structural coverage criterion PathCrawler was designed to satisfy, because the constraint solver is only called once for the initial test and then once for each node in the tree of feasible execution paths. Indeed, although constraint resolution is very fast most of the time, it is actually NP-complete and it is very difficult to know which constraint problem will take “too long” to resolve. This means that every time the constraint solver is called, there is a risk that it will run for “too long” and have to be interrupted by a timeout condition. This is why it is important to limit calls to the solver in order to speed up test generation. In structural testing, the minimum number of tests is defined by the structural coverage criterion and the tested function, so the only way to limit calls to the constraint solver is to limit the calls which do not contribute to this minimum number of tests. These are either calls which do not result in a test being generated, because the constraint problem is inconsistent (i.e. the path is infeasible), or else calls which generate a test which does not cover anything that has not already been covered by previous tests. PathCrawler limits the first category by always detecting infeasibility in the shortest prefix which is common to several infeasible paths. This paper is about limiting the second category.

PathCrawler generates tests to cover all feasible execution paths of functions coded in ANSI C (except functions containing certain constructions not treated yet, essentially pointer casts). However, for many functions the all-paths coverage criterion demands unmanageable numbers of tests. This phenomenon may be intrinsic to the structure of the tested function, for example if it contains numerous successive conditional instructions with few infeasible combinations (so the number of paths approaches 2 to the power of the number of conditional instructions), or loops containing conditional instructions (so that the number of potentially feasible paths is the number of paths through the loop body to the power of the number of iterations). In such cases, a branch-coverage criterion may be more appropriate.

However the number of paths to cover also depends on how paths are defined. Indeed, structural coverage criteria are not always defined very precisely in the literature, which can pose a problem for the practitioner. For instance, if called functions are treated as though they are in-lined in the code then they may cause a combinatorial explosion in the number of paths whereas coverage of the feasible paths through the tested function itself, without necessarily “covering” the called functions, may only demand a manageable number of tests. PathCrawler decomposes multiple conditions and then tries to cover the resulting expanded tree of feasible execution paths, which can also cause a combinatorial explosion in the number of “paths” whereas the number of paths through the different decisions of the multiple conditions may be manageable.

We would like to adapt PathCrawler to respect other control-flow-based structural coverage criteria in order to be applied to the programs for which the current criterion demands too many tests. As a first step, the alternative criteria which could be satisfied by a concolic generation strategy must be precisely defined and their interest to the user must be justified.

Moreover, we have also been investigating [6] the use of PathCrawler to generate tests to measure worst-case execution time (WCET). For this purpose, we have devised coverage criteria which exclude paths which, if certain hypotheses are respected, must have shorter execution times than the others. In these criteria, we may cover only the true branch of if-then-else structures with an empty else body or only the maximum number of iterations of loops with no condition in the loop body.

One way to respect coverage criteria other than all-paths would be to use the classic concolic path test generation strategy and just stop test generation when the criterion had been satisfied, e.g. in the case of branch coverage, when all branches had either been covered or proved unreachable. However, this is likely to be inefficient in the following sense. By *inefficient* test generation, we mean that many tests are generated and infeasible partial paths detected which are *redundant* in the sense that although they cover (or, in the case of infeasible partial paths, could have covered) new paths they do not increase coverage (resp. could not have increased coverage) as defined by the criterion in question, for example branch coverage. Redundant tests and redundant infeasible partial paths cost potentially expensive calls to the constraint solver and must be limited. Classic concolic test generation strategies explore the path tree “blindly” and so if, for example, only one branch remains to be covered, they may waste time exploring partial paths which are not even connected to that branch.

```
int g(int i, int x){
  if (i == x)
    return 2;
  else
    return (i*x)+1;
}

int f( int A[2], int e, int x) {
  int i, res ;
  res = 0 ;
  if((x < -1) || (x > 1)) {
   i = 0;
   while( (i < 2) && (res == 0)) {
     if( e == A[i] )
        res = g(i+1,x);
     else
        i++; } }
  if(res == 2)
    return 1;
  else
    return 0;
}
```

Figure 1.Source code of an example of a tested function, f.

In this paper, we present an abstraction of execution paths, “abstract paths”, which provides a conceptual framework to facilitate the definition and comparison of many different control-flow-based structural coverage criteria and of concolic generation strategies to efficiently satisfy these criteria. Structural coverage criteria are often said to be based on the program’s control-flow graph and abstract paths encapsulate parts of the control flow graph. However, the control-flow graph does not treat multiple conditions as we would like to and trees of abstract paths can also be compared to abstract syntax trees. In fact, abstract paths are a combination of the execution-path tree, the control-flow graph and the abstract syntax tree. We first used abstract paths for the WCET measurement criteria described in [6]. In the present paper, the idea of abstract paths is revised and generalised so that it can be used for other criteria. In the next section, we will first recall the tree of feasible execution paths explored by concolic test generation tools and introduce an example of a tested function and its feasible paths. In Section 3, we define abstract paths and in Section 4 we show how each of the criteria mentioned above can be defined in terms of these paths. In Section 5 we consider strategies to explore the abstract path graph in order to efficiently generate tests satisfying each criterion. Finally, we will discuss related work and future directions.

2. THE EXPANDED TREE

The classic concolic test generation strategy is an exhaustive exploration of a tree we will call the fully expanded tree of feasible execution paths (or expanded tree). To generate tests to satisfy other coverage criteria, the exploration of the whole of this tree should not usually be necessary. In order to discuss this further, we start by defining how the source code of the tested function is represented in this tree.

We suppose here that there are no system or library calls or GOTO instructions in the original source code and that it has been simplified so that it only contains if-then-else instructions with simple conditions, sequences of assignments and GOTO instructions added by the simplification. All conditional control instructions such as if-then-else, switch, while,… have been decomposed so that the only conditional instructions left have simple conditions containing no logical connectors such as && or side effects (assignments or function calls). Function calls have been replaced by assignments of the values of the effective parameters to the formal parameters, followed by the source code of the called function and then assignment of the return value.

What we call the *expanded tree* is in fact the tree of feasible execution paths through this simplified source code. It is composed of a root node, leaf nodes, conditional nodes and directed arcs between nodes. The *root node* represents the entry point of the tested function (which we suppose to be unique) and each *leaf node* represents an exit from the function. Each *conditional node* represents an if-then-else instruction with a simple condition. *Arcs* represent a truth value (true or false) and a (possibly empty) sequential block of unconditional instructions. There is a single arc (with value “true”) from the root to the first node and from the last node in each path to a leaf. Each conditional node has one arc entering it and either one or two arcs leaving it. Loops are unrolled. Each *path* from the root through connected arcs and nodes to a leaf represents a feasible execution path. A path which starts at the root and ends with an arc is called a *partial path*. If a conditional node in the expanded tree only has one arc leaving it, then the missing arc would be the final arc in an *infeasible partial path.*

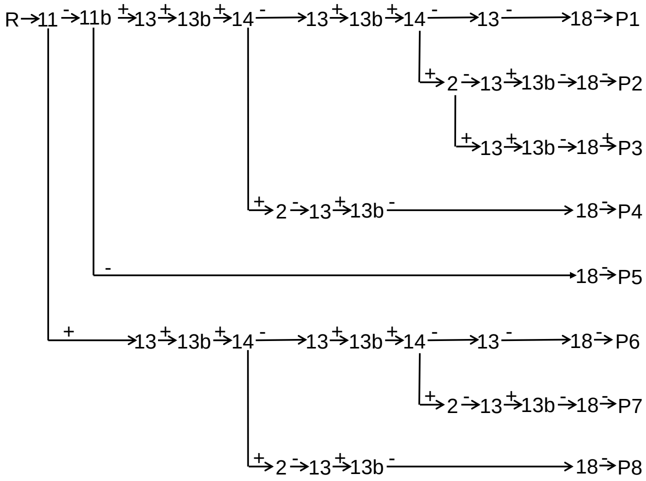

Figure 2. The expanded tree for f

As a running example of a tested function, we use the function "f" whose source code is displayed in Figure 1. In Figure 2 we show the expanded tree for this example. The root of this tree is labelled R and the leaves are labelled with the unique identifier of the feasible path (P1, P2,...) leading to this leaf. Each conditional node in this tree is labelled with line number of the conditional instruction in the source code that it represents, followed by the letter "b" if the node represents the second sub-condition in a multiple condition. The arcs are labelled with + for true and – for false but the assignments are not shown. In the following, we will denote the positive arc leaving a node labelled *n* by *n+* and the negative arc leaving the same node by *n-*.

The concolic exploration of the expanded tree starts with an arbitrary feasible path and then for each unexplored arc leaving a conditional node in this path it arbitrarily selects a feasible suffix (up to exit from the tested function) unless the partial path up to and including this unexplored arc is infeasible. The feasible suffix is explored in the same way. The arcs can be explored in any order. Figure 2 numbers the paths in an order which illustrates a possible concolic exploration of the arcs in depth-first order, supposing that the first, arbitrarily obtained, path is P1.

3. ABSTRACT PATHS

To introduce abstract paths, let us consider a test criterion which requires just the coverage of all feasible paths through the source code seen in a form in which called functions are not in-lined, nor multiple conditions decomposed. We will call this criterion *minimal-all-paths*. To respect minimal-all-paths for our example, it is not necessary to cover both P6 and P1, which only differ in the sub-conditions (11- and 11b+ in P1 and 11+ in P6) leading to the same positive decision for the multiple condition on line 11. We see that for this criterion, for each path (such as P1) in the expanded tree which is covered, there may be a set of feasible paths (such as P1 and P6) which are equivalent. Below, we will call this set of paths an abstract path for the minimal-all-paths criterion. Similarly, if the test criterion were coverage of all simple conditions, then after covering the first path, P1, the condition 14- is covered twice but not 14+. We would then try to cover either of the two partial paths ending in 14+ which is obtained by modifying one of the two prefixes of P1 which ends in 14- (and the next path covered would then be P2, P3 or P4). In this case, we would consider both partial paths as equivalent in spite of the fact that they have different loop iterations (sequences of 13+, 13b+ and 14+ or 14-).

Indeed, as shown by Figure 2, the same conditional instruction in the simplified code is usually represented by several different nodes in the expanded tree and two path fragments in this tree can be considered as equivalent, by certain test criteria and under certain conditions, if they start with the same conditional node, *s*, or different conditional nodes, *s1* and *s2*, representing the same conditional instruction and end with conditional nodes *t1* and *t2* both representing some other conditional instruction. For example, P1 and P6 in Figure 2 both contain a path fragment starting at the node, *s*, labelled 11 and ending at nodes *t1* and *t2* labelled 13. Such equivalent path fragments are produced when one of the following constructions is present in the un-simplified source code:

- If-then-else structures
- Function calls
- Loops
- Multiple conditions

Abstract path trees introduce into the expanded tree certain structural information found in the control flow graph or abstract syntax tree by grouping certain parts of the expanded tree into *abstract nodes* of the following types:

If-then-else This abstract node has one arc entering it, one arc leaving it and contains a conditional node and the two alternative path fragments.

Function call This has one arc entering it, one arc leaving it and contains all path fragments through the called function.

Loop This has one arc entering it, one arc leaving it and contains the conditional node of the loop head and all path fragments for one individual iteration

Logical Conjunction This has one arc entering it and two arcs, with two different truth values, leaving it. It contains the path fragment for the conjunction to be satisfied and the two path fragments for it to be false.

Logical Disjunction This also has one arc entering it and two arcs, with two different truth values, leaving it. It contains the two path fragments for the disjunction to be satisfied and the path fragment for it to be false.

Abstract nodes can be nested so the conditional node in an if-then-else can be a multiple condition and the alternative path fragments can contain other abstract nodes.

Conference'04, Month 1–2, 2004, City, State, Country.

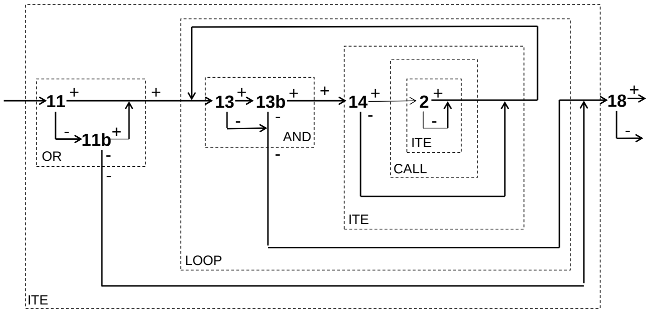

Figure 3. An abstract tree for f

Abstract trees are composed of the same root, leaves, arcs and conditional nodes as the expanded tree but they also contain abstract nodes. In Figure 3 we depict an abstract tree for our example, which contains all the types of abstract node mentioned above. As different path fragments within abstract nodes can lead to a common node, they are not depicted as a tree but as paths through a graph which is similar to a fragment of the control flow graph. Note that within an abstract node, one node, such as the node labelled 14+ in Figure 3, may represent several nodes in the expanded tree (in this case, all the nodes labelled 14+ in the expanded tree).

Abstract paths are paths from the root to a leaf of an abstract path tree. Note that when an abstract path goes through a conditional abstract node (i.e. logical conjunction or disjunction), it follows just one arc out (true or false), as in the case of non-abstract conditional nodes in concrete paths. Each abstract path represents a set of concrete paths in the expanded tree. From now on, we will refer to paths in the expanded tree as *concrete paths*. The abstract tree depicted in Figure 3 contains two abstract paths, one representing all feasible paths whose final arc represents the truth of the conditional instruction at line 18, and the other representing all the feasible paths for which this condition is false. As abstract nodes can be nested, an abstract path can contain path fragments which themselves traverse abstract nodes but we will refer to these as *abstract path fragments* and not abstract paths.

Note that the expanded tree only represents feasible paths but the feasibility of a path fragment in an abstract node may depend on the partial path leading to the abstract node if this partial path contains other abstract nodes. We consider that each arc in an abstract tree (including the arcs in the abstract nodes) is reachable, i.e. is present in at least one concrete path.

An *abstract partial path* is a path from the root to an arc which leaves a non-abstract conditional node. Note that if the abstract partial path ends in an arc which is contained in an abstract node, *n*, (or in nested abstract nodes *n1*, *n2*,…) then *n* (resp. *n1*, *n2*,…) must be expanded in the abstract partial path.

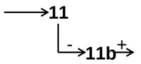

Figure 4. The abstract partial path to arc 11b+

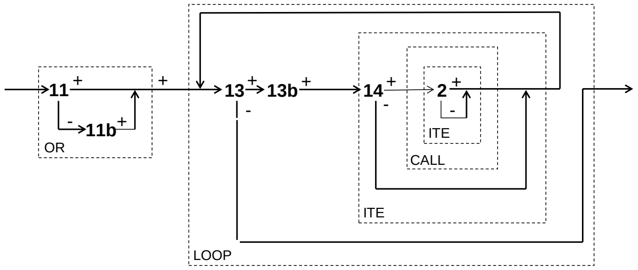

Figure 5. The abstract partial path to arc 13-

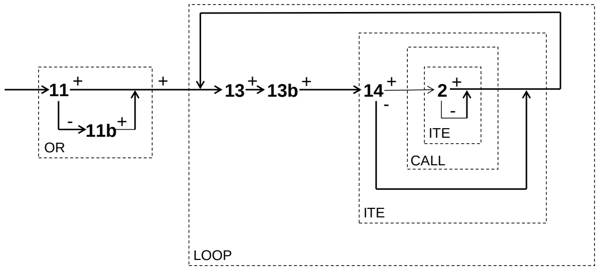

Figure 6. Abstract partial path to arcs 13+, 13b+, 14+, 2+, 2-

For example, Figure 4 depicts the abstract partial path to the arc labelled 11b+ in the abstract tree depicted in Figure 3, which corresponds to a single concrete partial path because the abstract logical disjunction node has been expanded. In the case of an arc contained in an abstract loop node, the abstract partial path to the arc may represent concrete paths with several different iterations before the final arc is reached, and with different numbers of iterations before the final arc. This is why we depict such abstract partial paths as in Figure 5, which shows the abstract partial path leading to the arc 13- in the abstract tree depicted in Figure 3. The abstract partial paths leading in Figure 3 to the arcs 13+, 13b+, 14+, 14-, 2+ and 2- are all depicted by Figure 6.

4. USE OF ABSTRACT TREES TO DEFINE COVERAGE CRITERIA

Let us now see how different control-flow-based structural coverage criteria can be defined in terms of abstract trees containing different abstract nodes.

The *minimal-all-paths* criterion was already discussed above. It corresponds to the coverage of all abstract paths in an abstract tree in which only function calls and multiple conditions are encapsulated in abstract nodes. If-then-else structures and loops are left in their expanded form in this tree. Figure 7 depicts this abstract tree for our example.

If the tested function has too many abstract paths even when the function calls and multiple conditions are abstracted then the abstraction of loops can be considered. One criterion commonly used in this case is *k*-path, in which *k* is a small integer constant fixed by the user and only the only paths covered are those with up to *k* iterations of any loop with a variable number of iterations. However, this criterion does not take into account the branches within each iteration and if any paths are only feasible when a path contains more than k iterations of one of these loops, then these paths will not be tested. Abstract loop nodes allow other, more justifiable, criteria to be defined but we do not have space to discuss them here.

If even the abstraction of function calls, multiple conditions and loops leaves too many paths to cover, then the user can decide to cover either all simple conditions, including the sub-conditions of multiple conditions, or just all decisions. We call *minimal-all-conditions* the coverage of all simple conditions in the tested function (but not all conditions in any called functions). This is defined using an abstract tree in which if-then-else structures, function calls, loops and multiple conditions are all encapsulated in abstract nodes (i.e. for our example, the tree depicted in Figure 3). The criterion corresponds to the coverage of all abstract partial paths in which the last arc is not contained in an abstract function call node (see Figures 4 – 6 for our example).

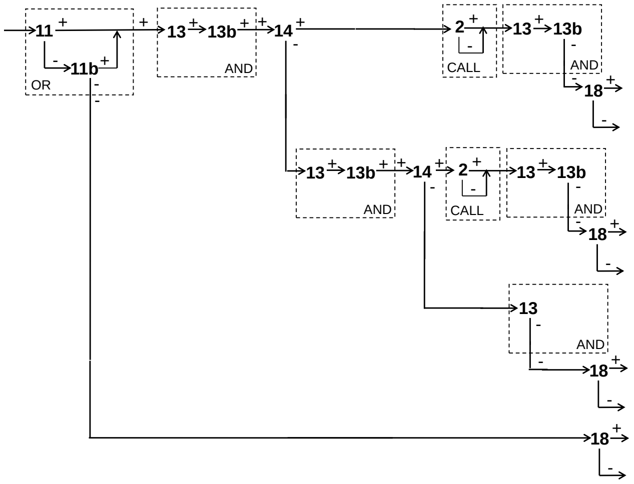

Figure 7. The abstract tree for the minimal-all-paths criterion

We will call the next criterion *minimal-all-decisions*. It is coverage of all branches appearing in the tested function's original source code, in which multiple conditions are not decomposed and function calls are not in-lined. This criterion corresponds to the coverage, in the same abstract tree as the previous criterion, of all abstract partial paths in which the last arc leaves either a simple condition which is not contained in an abstract function call or multiple condition node or else an abstract multiple condition node which is not contained in another abstract multiple condition node, nor in an abstract function call node.

The final two criteria we define here result from our previous work [6] on the generation of tests for measuring the worst-case execution time. They are based on coverage of all paths but differ from the previous criteria in that they introduce a partial order between paths. The first is called *empty-else*: do not cover a path, *p1*, if there is another feasible path, *p2,* such that the only difference between *p1* and *p2* is that in one or more if-then-else structures *p1* follows an empty path fragment (containing no instructions) and *p2* follows a non-empty path fragment. This criterion is defined using an abstract tree in which only if-then-else structures with empty else bodies (ITEE structures, such as the outermost ITE in Figure 3) are encapsulated in abstract nodes. The second criterion is called *max-iterations*: do not cover a path, *p1*, if there is another feasible path, *p2,* such that the only difference between *p1* and *p2* is that in one or more loops in which all iterations are identical, *p1* executes fewer iterations than *p2*. This criterion is defined using an abstract tree in which only loops are encapsulated in abstract nodes. In the trees for these two criteria, there is a partial order on the concrete paths belonging to each abstract path. For example, an abstract path which traverses one abstract ITEE node represents both the concrete paths which take the empty path fragment of this ITEE and those that take a non-empty path fragment. However, if there are several non-empty path fragments through the same ITEE, or if there are several ITEEs in the same abstract path, then the different concrete paths may not be comparable because the order is only partial. These last two criteria are satisfied if, for each abstract path in the tree, we cover all concrete paths which are maximal in the corresponding partial order over this abstract path.

5. USE OF ABSTRACT TREES TO DEFINE TEST GENERATION STRATEGIES

Now let us show how abstract trees can be used to define concolic strategies to generate tests to satisfy the first two criteria above without exploring as many feasible paths or infeasible partial paths as the classic depth-first concolic strategy.

As mentioned in the Introduction, PathCrawler was designed to generate a set of tests to guarantee 100% satisfaction of a structural coverage criterion (in the cases where the constraint solver does not timeout during test case generation). This enables the user to give a precise measure of confidence in a program for which all tests in the set succeed. PathCrawler can therefore be used not only for debugging but as part of a more formal test process. Part of the service to the user provided by tools such as PathCrawler is the guarantee that all paths or branches which are not covered by the test set are infeasible resp. unreachable (in the case of constraint resolution timeout, the path prefixes are indicated to the user, who must check their feasibility by hand). The test generation strategy must ensure this service, as well as the generation of the test set. Concolic tools such as [3], which use over-approximate path predicates in order to be able to treat all programs, do not necessarily attempt to provide this service and so would choose slightly different test generation strategies to those proposed below.

Note that our proposed strategies also take into account one reason why PathCrawler is efficient: because it is implemented using constraint logic programming, depth-first search makes the most of Prolog's built-in backtrack mechanism to store successive states of the constraint store in a stack instead of recalculating them on backtrack.

In Figures 8 and 9, we illustrate the first two strategies on our example by giving the covered paths (labelled P or R), and infeasible partial paths (labelled I) explored, still assuming the arbitrary first path is P1. Each path or partial path is denoted by its successive arcs and each path fragment which is inside an abstract node is enclosed in brackets, as follows: ( ) for a loop, < > for AND, « » for OR, [ ] for ITE and {} for CALL.

As explained in the previous section, the strategies concern coverage of abstract paths in the case of path-based strategies such as minimal-all-paths and empty-else and coverage of abstract partial paths in the case of arc-based strategies such as minimal-all-conditions and minimal-all-decisions. We will use the term *(partial) paths* to mean paths in the case of path-based strategies and partial paths in the case of arc-based strategies.

A concolic strategy consists in first defining the order in which to inspect the arcs in the concrete paths already covered. Then, for each arc, *a*, which is in a concrete path, *P*, which has already been covered, the strategy must define whether and how to explore the alternative arc, *a′*. Let *PP* be the concrete prefix of *a* in *P* and *PP′* be the concrete partial path formed by adding *a′* to the end of *PP*. The strategies defined using abstract trees are based on the following principles:

1. If *PP′* belongs to an abstract partial path which has not yet been covered, then try to generate any test covering *PP′*
2. If *PP′* is only a prefix of concrete (partial) paths which belong to already covered abstract (partial) paths, then there is no need to cover *PP′*.
3. If *PP′* is a prefix of at least one concrete (partial) path, *PPP′*, belonging to an abstract (partial) path that has not yet been covered, then decide how to try and cover *PPP′* and all other concrete (partial) paths of which *PP′* is a prefix and which belong to uncovered abstract (partial) paths.

Point (2) above defines when *PP′* does not need to be covered and points (1) and (3) concern different ways to try and cover *PP′*.

We propose to divide arc-based strategies into two passes. In a first breadth-first traversal, only the partial paths corresponding to point (1) above are explored. Then the still uncovered partial paths are reviewed and, in a second pass, those corresponding to point (3) above are explored: all concrete partial paths belonging to each uncovered abstract partial path are systematically explored until either a test to cover the abstract partial path is found or all the concrete partial paths belonging to it have been proved infeasible. The justification for breadth-first search and this two-phase scheme is opportunistic: in covering *PP′*, we cover a whole new concrete path, *P′*, of which *PP′* is a prefix, and *P′* may contain other uncovered arcs as well as *a′*. (or in other words, *P′* may have other prefixes which belong to different abstract partial paths). If we explore covered paths breadth-first then we first explore the alternatives of arcs at the beginning of paths. These are more likely to have long suffixes, thereby increasing the

```
[<11- 11b+>+ («13+ 13b+ »+ [14-      ] «13+ 13b+»+  [14-      ] «13-      »- )] 18-  : P1
[<11- 11b+>+ («13+ 13b+ »+ [14-      ] «13+ 13b+»+  [14-      ] «13-      »- )] 18+  : I1
 <11- 11b+>+  «13+ 13b+ »+ [14-      ] «13+ 13b+»+  [14-      ]  13+                   : I2
[<11- 11b+>+ («13+ 13b+ »+ [14-      ] «13+ 13b+»+  [14+ {-2} ] «13+ 13b- »- )] 18-  : P2
[<11- 11b+>+ («13+ 13b+ »+ [14-      ] «13+ 13b+»+  [14+ {-2} ] «13+ 13b- »- )] 18+  : I3
 <11- 11b+>+  «13+ 13b+ »+ [14-      ] «13+ 13b+»+  [14+ {-2} ] «13+ 13b+»+          : I4
[<11- 11b+>+ («13+ 13b+ »+ [14-      ] «13+ 13b+»+  [14+ {-2} ] «13-      »- )] 18+  : I5 (suffix18+)
[<11- 11b+>+ («13+ 13b+ »+ [14-      ] «13+ 13b+»+  [14+ {+2} ] «13+ 13b- »- )] 18+  : P3
[<11- 11b+>+ («13+ 13b+ »+ [14-      ] «13+ 13b+»+  [14+ {+2} ] «13+ 13b- »- )] 18-  : I not explored
 <11- 11b+>+  «13+ 13b+ »+ [14-      ] «13+ 13b+»+  [14+ {+2} ] «13+ 13b+»+          : I6
 <11- 11b+>+  «13+ 13b+ »+ [14-      ] «13+ 13b+»+  [14+ {+2} ] «13-      »-          : I not explored
 <11- 11b+>+  «13+ 13b+ »+ [14-      ] «13+ 13b- »-                                  : I7
 <11- 11b+>+  «13+ 13b+ »+ [14-      ] «13-      »-                                  : I8
[<11- 11b+>+ («13+ 13b+ »+ [14+ {-2} ] «13+ 13b- »-                         )] 18-  : P4
[<11- 11b+>+ («13+ 13b+ »+ [14+ {-2} ] «13+ 13b- »-                         )] 18+  : I9
 <11- 11b+>+  «13+ 13b+ »+ [14+ {-2} ] «13+ 13b+»+                                  : I10
[<11- 11b+>+ («13+ 13b+ »+ [14+ {-2} ] «13-      »-                         )] 18+  : I11 (suffix18+)
 <11- 11b+>+  «13+ 13b+ »+ [14+ {+2} ]                                               : I12
 <11- 11b+>+  «13+ 13b-  »-                                                          : I13
 <11- 11b+>+  «13-       »-                                                          : I14
[<11- 11b- >-                                                                  ] 18-  : P5
[<11- 11b- >-                                                                  ] 18+  : I15
[<11+     >+ («13+ 13b+ »+ [14-      ] «13+ 13b+»+  [14-      ] «13-      »- )] 18-  : R1 (P6~P1)
[<11+     >+ («13+ 13b+ »+ [14-      ] «13+ 13b+»+  [14-      ] «13-      »- )] 18+  : I16
 <11+     >+  «13+ 13b+ »+ [14-      ] «13+ 13b+»+  [14-      ]  13+                   : I17
[<11+     >+ («13+ 13b+ »+ [14-      ] «13+ 13b+»+  [14+ {-2} ] «13+ 13b- »- )] 18-  : R2 (P7~P2)
[<11+     >+ («13+ 13b+ »+ [14-      ] «13+ 13b+»+  [14+ {-2} ] «13+ 13b- »- )] 18+  : I not explored
 <11+     >+  «13+ 13b+ »+ [14-      ] «13+ 13b+»+  [14+ {-2} ] «13+ 13b+»+          : I18
 <11+     >+  «13+ 13b+ »+ [14-      ] «13+ 13b+»+  [14+ {-2} ] «13-      »-          : I not explored
 <11+     >+  «13+ 13b+ »+ [14-      ] «13+ 13b+»+  [14+ {+2} ] «13+ 13b+»+          : I19 (suffix 13+ 13b+)
 <11+     >+  «13+ 13b+ »+ [14-      ] «13+ 13b- »-                                  : I20
 <11+     >+  «13+ 13b+ »+ [14-      ] «13-      »-                                  : I21
[<11+     >+ ( «13+ 13b+ »+ [14+ {-2} ] «13+ 13b- »-                        )] 18-  : R3 (P8~P4)
[<11+     >+ («13+ 13b+ »+ [14+ {-2} ] «13+ 13b- »-                         )] 18+  : I22
 <11+     >+  «13+ 13b+ »+ [14+ {-2} ] «13+ 13b+»+                                  : I23
[<11+     >+ («13+ 13b+ »+ [14+ {-2} ] «13-      »-                         )] 18+  : I24 (suffix 18+)
 <11+     >+  «13+ 13b+ »+ [14+ {+2} ]                                               : I25
 <11+     >+  «13+ 13b-  »-                                                          : I26
 <11+     >+  «13-       »-                                                          : I27
```

Figure 8. Minimal-all-paths strategy illustrated on our example

chances of quickly finding other uncovered arcs.

Conversely, only one path can be covered at a time, whether it is abstract or concrete, so for path-based strategies we propose to take advantage of the efficiency of depth-first search.

Now let us see how this general scheme can be instantiated for the first two criteria. Note that the precise details of the exploration of the arcs corresponding to point (3) above are just proposed as an example; they could be explored in other ways.

5.1 Minimal-all-paths

This strategy is illustrated in Figure 8, which maintains the same order of paths as in Figure 2 because this strategy is also based on depth-first search. The infeasible partial paths in Figure 8 are also traversed in the same order as they would be in exhaustive depth-first search of the expanded tree, but in this strategy some partial paths do not need to be explored, whilst some only need to be explored with a particular suffix. Finally, in order to explore the necessary concrete suffixes of *PP′*, this strategy starts by trying to cover the shortest common suffix. This risks generation of redundant tests (labelled with the letter R in Figure 8), but avoids repeated failures.

For this criterion, if *a* is not contained in an abstract node, then the strategy always tries to generate a test covering *PP′*. In our example, this is the case for the last arc in P1, which is 18-, but for which *PP′* (I1 in Figure 8) is found to be infeasible.

If *a* is contained in a multiple condition node *n* with decision *d* and replacing *a* with *a′* can lead to the opposite decision, *d′*, then the strategy must check whether this is feasible. However, if replacing *a* with *a′* can also lead to decision *d*, and the resulting partial path could be an alternative way to cover an abstract partial path that has already been unsuccessfully tried, then this must also be considered. This is the case in our example for the next *a* considered, 13-, which belongs to the multiple condition at line 13. In this case *a′* is 13+ and with suffix 13b+ it would change the decision at line 13. However, with suffix 13b- 18+ it would enable the abstract partial path of I1 to be covered. Rather than enumerating both suffixes, the strategy first tries to cover the shortest suffix of *PP′* common to both. This is empty so the strategy just tries to cover *PP′*, which is I2 in Figure 8, without success.

At the end of this example, 4 fewer infeasible partial paths have been explored than with an exhaustive exploration of the expanded tree. If we had tried to just stop an exhaustive exploration once this criterion were satisfied, we would not have saved any exploration, because the infeasibility of some of the abstract paths is only proved at the end of the exploration.

5.2 Minimal-all-conditions

For this criterion, if *a* is not contained in a called function and does not already appear negated in one of the paths covered so far, then the strategy always tries to generate a test covering *PP′*. This is the case for the last arc in P1, which is 18-, but for which *PP′* (I1 in Figure 8) is found to be infeasible. The first phase explores all these cases breadth-first, as illustrated in Figure 9 until the failure to negate 18+ in I3. The second phase then considers all arcs *a* contained in a called function or which already appear negated in one of the paths covered so far but for which *a′* could be an alternative way to cover an abstract partial path (in this case, the abstract partial path leading to 18+).

At the end of this example, 10 infeasible partial paths and 1 redundant feasible path (a total of 11) have been explored whereas if exhaustive search of the expanded tree had just been stopped once this criterion were satisfied, after covering P6, then 17 infeasible partial paths and 1 redundant feasible path (a total of 18) would have been explored.

6. CONCLUSION

Abstract nodes add structural information to the fully expanded tree of feasible execution paths used by concolic test generation tools. This structural information is also present in the control flow graph and the abstract syntax tree and we could have used these to define different criteria and the corresponding test generation strategies. However, the abstract paths presented in this paper are particularly well adapted to the precise definition of structural control-flow-based test criterion and efficient concolic test generation strategies.

Indeed, concolic test generation tools can only become really useful if they can be applied to a reasonably large class of programs. This implies that efficient variants of their classic exploration strategy must be found which still retain the advantages of the concolic approach.

```
[ <11- 11b+>+  «13+ 13b+»+ [14-      ] «13+ 13b+»+ [14-      ] «13-       »- )] 18-  : P1
[ <11+     >+  «13+ 13b+»+ [14-      ] «13+ 13b+»+ [14-      ] «13-       »- )] 18-  : P2
[ <11- 11b- >-                                                                ] 18-  : P3
  <11- 11b+>+  «13+ 13b- »-                                                          : I1
  <11+     >+  «13+ 13b- »-                                                          : I2
[ <11+     >+ («13+ 13b+»+ [14+ {-2}] «13+ 13b- »-                           )] 18-  : P4
[ <11+     >+ («13+ 13b+»+ [14+ {-2}] «13+ 13b- »-                           )] 18+  : I3
[ <11- 11b- >-                                                                ] 18+  : I4
[ <11+     >+  «13+ 13b+»+ [14-      ] «13+ 13b+»+ [14-      ] «13-       »- )] 18+  : I5
[ <11- 11b+>+  «13+ 13b+»+ [14-      ] «13+ 13b+»+ [14-      ] «13-       »- )] 18+  : I6
  <11- 11b+>+  «13+ 13b+»+ [14-      ] «13+ 13b+»+ [14-      ]  13+                  : I7
[ <11- 11b+>+ («13+ 13b+»+ [14-      ] «13+ 13b+»+ [14+ {-2} ] «13+ 13b- »- )] 18-  : R1
[ <11- 11b+>+ («13+ 13b+»+ [14-      ] «13+ 13b+»+ [14+ {-2} ] «13+ 13b- »- )] 18+  : I8
[ <11- 11b+>+ («13+ 13b+»+ [14-      ] «13+ 13b+»+ [14+ {-2} ] «13+ 13b+ »+)] 18+  : I9
[ <11- 11b+>+  «13+ 13b+»+ [14-      ] «13+ 13b+»+ [14+ {-2} ] «13-       »- )] 18+  : I10
[ <11- 11b+>+ («13+ 13b+»+ [14-      ] «13+ 13b+»+ [14+ {+2} ] «13+ 13b-»-  )] 18+  : P5
```

Figure 8. Minimal-all-conditions strategy illustrated on our example

There has been much recent work on this subject, but each time from the point of view of a particular cause of "path explosion". In [10], it is function calls which are abstracted and the term "abstract path" is also used. However, it is defined as the path described by a predicate in which the input-output relations of called functions are not known. In our terms this is equivalent to a path through the tested function in which function calls are encapsulated in an abstract node, but before any paths through the abstract node are known. The proposed strategy consists in first exploring concolically all "abstract paths" which are feasible when the function calls are replaced by stubs which can return any value. Then the real called functions are put back in place of the stubs and for each of these "abstract paths", and for each function call, different concrete paths through the real called function are explored concolically until one is found that is consistent with the rest of the "abstract path". [9] does not use the term "abstract path" but it also manipulates path predicates in which we can consider that the path through called functions is abstracted. However the predicates of [9] only characterise abstract paths for which at least one path through each called function is known. The strategy proposed in [9] is more efficient than that of [10]. It memorises the predicates of all known paths through called functions, along with the calling context predicates. This means it only needs to concolically explore alternative paths through a called function if it fails to construct a feasible predicate by inserting a previously memorised called function path predicate into the predicate of the path through the tested function. In our own previous work based on formal specifications of library functions [8], we also proposed a test generation algorithm to avoid unnecessary exploration of these specifications.

In other work, the focus is on adapting the strategies of particular tools in order to obtain more efficient statement or branch coverage, although their aim may not be to completely satisfy a formally defined criterion and so they may not be concerned, as we are above, about demonstrating the unreachability of the uncovered statements or branches.

In [4], heuristics are proposed to decide whether to explore a branch which has already been covered, but with a different prefix. However, the first heuristic proposed is the connection to an uncovered branch, rather as in our minimal-all-conditions strategy of Section 5 above, and [4] discusses the most efficient way to compute this information. In [11] the goal is to quickly cover most reachable statements and the program structure in terms of "building blocks such as methods and loops" is taken into account in the definition of a fair choice between unexplored branches. In [7], when concolic depth-first exploration arrives at a "context-sensitive program point" (such as an exit from an ITE) the exploration of the part of the execution-path tree which is rooted at this program point is used to discover which variables are live at this point. These variables are memorised along with their intersection with the current program state (path constraint and concrete memory state). On the next traversal of the same program point, this memorised dependence data is retrieved and used to decide whether exploration of a different path through the ITE will only result in covering the same suffixes (or in our terms, the same abstract paths).

In conclusion, previous work may well achieve greater test generation efficiency than the strategies proposed here but it is mostly specialised to a particular criterion, which may not be very precisely defined. Only our framework aims to facilitate the precise definition and comparison of different criteria and concolic generation strategies for control-flow-based structural testing.

Indeed, the test generation strategies based on abstract trees which we propose here just allow control dependences to be taken into account in order to save some unnecessary constraint resolution. The next step is to take data dependences into account to save even more unnecessary exploration and we believe that abstract trees will also provide a convenient framework in which to implement this. This will be the focus of our future work.

7. REFERENCES

[1] B. Marre, P. Mouy and N. Williams, "On-the-Fly Generation of K-Path Tests for C Functions", 19th IEEE Intnl. Conf. on Automated Software Engineering (ASE 2004), September 2004, Linz, Austria.

[2] N. Williams, B. Marre, P. Mouy and M. Roger, PathCrawler: "Automatic generation of path tests by combining static and dynamic analysis", In Proc. EDCC-5, Budapest, April 2005.

[3] K. Sen, D. Marinov and G. Agha "CUTE: a concolic unit testing engine for C", In ESEC/FSE'O5, pp 263-272, Lisbon, Portugal, September 2005

[4] S. Bardin and P. Herrmann, "Structural testing of executables" in Proc. ICST'08, pp 22-31, Lillehammer, Norway, April 2008.

[5] C.Cadar, V.Ganesh, P.M.Pawlowski, D.L.Dill, and D.R.Engler, "Exe: automatically generating inputs of death", In Proc. ACM Conference on Computer and Communications Security, 2006.

[6] Nicky Williams, Muriel Roger, "Test Generation Strategies to Measure Worst-Case Execution Time", AST'09, Vancouver, May 2009.

[7] P. Boonstoppel, Cristian Cadar, Dawson Engler, "RWSet: Attacking path explosion in constraint-based test generation", In Proc. TACAS 2008, Budapest, Hungary, March-April 2008

[8] P. Mouy, B. Marre, N.Williams and P. Le Gall, "Generation of all-paths unit test with function calls", In Proc. ICST'08, Lillehammer, Norway, 2008.

[9] S. Anand, P. Godefroid, N. Tillmann, "Demand-Driven Compositional Symbolic Execution", In Proc. TACAS 2008, Budapest, Hungary, 2008.

[10] R. Majumdar and K. Sen, "LATEST: Lazy dynamic test input generation". Technical Report UCB/EECS-2007-36, EECS Department, University of California, Berkeley, 2007.

[11] Nikolai Tillmann, Jonathan de Halleux, "Pex – White Box Test Generation for .NET", Proc. Of TAP 2008, LNCS, vol. 4966, pages 134-153, April 2008.