Generation of all-paths unit test with function calls

Patricia Mouy, Bruno Marre, Nicky Williams
CEA/LIST, LSL, 91191 Gif Sur Yvette Cx, France
firstname.lastname@cea.fr

Pascale Le Gall
Ecole Centrale Paris, Laboratoire MAS
Grande voie des Vignes, 92295 Chatenay Malabry, France
pascale.legall@epigenomique.genopole.fr

Abstract

Structural testing is usually restricted to unit tests and based on some clear definition of source code coverage. In particular, the all-paths criterion, which requires at least one test-case per feasible path of the function under test, is recognised as offering a high level of software reliability. This paper deals with the difficulties of using structural unit testing to test functions which call other functions. To limit the resulting combinatorial explosion in the number of paths, we choose to abstract the called functions by their specification. We incorporate the functional information on the called functions within the structural information on the function under test, given as a control flow graph (CFG). This representation combining functional and structural descriptions may be viewed as an extension of the classic CFG and allows us to characterise test selection criteria ensuring the coverage of the source code of the function under test.

Two new criteria will be proposed. The first criterion corresponds to the coverage of all the paths of this new representation, including all the paths arising from the functional description of the called functions. The second criterion covers all the feasible paths of the function under test only. We describe how we automate test-data generation with respect to such grey-box (combinations of black-box and white-box) test selection strategies, and we apply the resulting extension of our PathCrawler tool to examples coded in the C language.

1 Introduction

Structural testing is popular because the coverage of the source code can be clearly defined and quantified. However, the confidence in the code which can be inferred from a successful test run depends on the coverage criteria and on the oracle used to evaluate the test results. Even assuming a perfect oracle, 100% coverage of reachable instructions or branches only guarantees the detection of errors which are always provoked, whatever the context of execution of the instruction or branch. However, in the case of full coverage of feasible execution paths, all instructions and branches are tested in all possible execution contexts.

Even for unit testing, full path coverage is often assumed to be an unrealistic goal because of the number of tests required. This is particularly true when it is not automated. We have developed the PathCrawler tool which automatically generates test inputs for 100% coverage of feasible paths for unit testing of C source code. With an automatic oracle, PathCrawler makes fully automatic unit testing possible for many C functions. For the treatment of loops with a variable number of iterations, the all-path criterion can be restricted to a variant : the k-path criterion (also known as ct-coverage [BM93]) where the number of iterations in loops is bounded by a given parameter k. However, path coverage must be carefully defined when the function under test calls other functions. Up until now, PathCrawler has inlined the source code of called functions, risking a combinatorial explosion in the number of paths and an unnecessarily large number of tests if we consider that the all-feasible-paths criterion only requires coverage of all paths in the function under test itself.

For function calls, two methods are usually used. The inlining method consists of including the source code of the called functions in the source code of the function under test so that the test criterion is applied to it as well. This method amplifies the problem of the combinatorial explosion of paths by combining the number of paths of the called functions with the number of paths of the function under test. The second method consists of replacing the called functions with specialised modules, called stubs, built manually in an ad-hoc way and often incomplete. In conclusion, the usual ways to treat function calls cannot be used for the automatic generation of unit tests. In this paper, we address this limitation of structural testing. Our objective is to ensure the preservation of full coverage of paths in the calling function, while limiting the combinatorial explosion of tested paths.

Example 1.1 *This example serves us as a motivating example and is inspired by a technique commonly used in embedded software for the approximation of a continuous function (linear approximation). It is taken from a real-world industrial application, typical of critical embedded-systems code. Consider the following example of source code. The function* `apply_f` *is computing an approximation of a function f and contains two function calls of the function* `get_inter`*. More precisely, the calling function* `apply_f` *contains 6 different execution paths, 5 of which contain a call to the function* `get_inter`*.*

```c
const int n=9;
const int t[9]={-15,-5,-3,-1,0,1,3,5,15};
const int f[9]={5,4,3,2,1,2,3,4,5};

/* CALLED FUNCTION returns -1, -2 or value in [0,n-2]*/
int get_inter(int val){
 int r; int j;
 if (val < t[0])
  r = -2;
 else
  if (val>=t[n-1])
   r = -1;
  else
   for(j=0;j<n-1;j++){
    if ((val>=t[j])&&(val<t[j+1])){
     r = j;
     break;}
   }
 return (r);
}

/* CALLING FUNCTION */
int apply_f(int x, int mode){
 int i; int ret;
 if (mode)
  if !((x>=0)&&(x<5))
   ret = -300;
  else {
   i = get_inter(x); /*FUNCTION CALL (x:[0,5[)*/
   if (i<0)
    ret = -100;/*DEAD CODE : the called function
    returns 4,5 or 6 with the previous calling context */
   else
    ret = f[i] * 2; }
 else {
  i = get_inter(x); /*FUNCTION CALL (x:int)*/
  if (i<0)
   ret = -100;
  else {
   if (i > n-1)
    ret = -200;/*DEAD CODE : the called function
    cannot return value > n-1 */
   else
    ret = f[i];}
 }
 return ret;
}
```

Although it is simple, the source code of the called function `get_inter` results in a combinatorial explosion in the number of paths in the calling function. This is because each path in the calling function is duplicated for each possible number of iterations of the loop in the called function, i.e. for each element of the constant array `t`.

In this example, in each duplicated path of the calling function `apply_f`, a different element of the approximated function, f, is applied but the calculation does not otherwise change. In other words, the duplicated paths exercise different constant data values and not different algorithms. In this case, the real differences in the algorithm correspond to the different paths in the calling function `apply_f` following the calls to `get_inter` (2 paths following the first call and 3 paths following the second call). There is also an execution path without a function call. Only these paths need to be covered for the coverage of all the feasible paths of the calling function `get_inter`.

Our method is based on a formal specification of the called function. We assume that either the source code of the called function is available and has already been independently validated or else the called function is library or off-the-shelf software component (COTS) for which the source code is not available but there is a detailed description of the functionality and restrictions on usage. We believe that in such cases, in a context of automated unit testing, users are prepared to formalise the specifications of the called functions. Either the called functions must in any case be independently tested or proved, in which case formal specifications would be very useful anyway, or else they are documented in a form which is easy to transcribe to formal specifications. We propose a specification language which uses the same function names and types as the C code and structures the specifications into pre/post conditions. It is similar to the usual languages used for defining assertions in source code (such as JML).

This work is an example of grey-box test selection strategy that advantageously combines white box (structural) and black-box (functional) strategies in order to achieve automation of unit testing.

In Section 2, we discuss related work and explain our unit test method for PathCrawler without function calls. In the next section, we present our novel treatment of function calls. Next, we explain the two criteria defined for unit testing with function calls. In Section 5, we present our first results obtained with the prototype implementation and we discuss this approach. Finally, we conclude with a brief wrap-up and some perspectives.

2 PathCrawler overview and Related works

Our method [WMM03], [WMMR05], [WMM04] named PathCrawler is a "path testing" method for sequential programs coded in an imperative language. Our test-data generator prototype, also called PathCrawler, treats

ANSI C programs.

Like the dynamic approaches [FK96], [Kor96], [GN97], [GMS99] and [MM98] to test data generation, our method is based on dynamic analysis. Most dynamic approaches use heuristic function minimisation. The iterative relaxation method in [GMS99] uses a linear approximation of the path predicate but requires many executions of the function to derive an input for a given path. We use constraint logic programming (CLP) to solve a partial path predicate. In this way, our approach resembles [GBR00] and [SD01], which are also based on constraint resolution. However, we suffer neither from the complexity of their purely symbolic approaches, nor from the number of executions demanded by the dynamic approaches.

Unlike all the preceding approaches, PathCrawler aims to cover all execution paths and not a particular path. Indeed, PathCrawler was one of the first methods (after [PM87]) to be based on the modification of a predicate of a previously covered path. These methods now include CUTE[SMA05], DART[GKS05] and EXE[CGP+06], called “concolic” because they are based on a collaboration of concrete instrumented executions and symbolic execution analysis. For the instrumentation step, all the tools use the static analysis tool CIL [Lan06]. Each path predicate built from an instrumented execution is then analysed to derive data inputs exercising an uncovered path in an attempt to cover all feasible execution paths. PathCrawler, DART and CUTE perform a depth-first exploration of the CFG of the function, while EXE performs a kind of breadth-first exploration based on a system of forked executions at each branching node of the CFG.

While PathCrawler is a structural test-data generation tool (for the all-paths criterion), these other concolic tools are presented as bug-finding tools, exploring execution paths in order to exhibit run-time errors (e.g. divide by zero), security flaws (e.g. buffer overflows) or violations of user assertions. Since they do not ensure the coverage of a testing criterion, the resolution procedures used for path predicate solving in DART, CUTE or EXE do not need to be complete and can be limited to linear constraints (as is the case for DART and CUTE). The completeness (w.r.t. the test criterion) of PathCrawler comes both from its test-data selection strategy and its constraint solver which also handles non-linear constraints [WMMR05]. However, this theoretical completeness holds only in the absence of a time-out during constraint resolution. Solving constraints over finite domains and finite path predicates is NP-complete in the worst case but in practise, our test selection strategy and constraint solving heuristics alleviate this problem. In PathCrawler, test selection and constraint solving are implemented in the ECLiPSe constraint logic programming environment [WNJ97]. Our strategy of constraint resolution uses a randomised labelling strategy similar to the one used in [MA00].

One major difficulty in program analysis is the handling of alias relations. This problem is simplified when analysing one unrolled path at a time as is done in all these adaptive methods. A common restriction is to suppose that there is no alias between fields of data-structure inputs unless explicitly specified by the user. Aliases then pose the following two problems : 1) when translating branch conditions into constraints on input values, intermediate assignments which use a different alias must be recognised 2) creating a test-case which exercises a branch condition which can only be satisfied in the case of an alias between different fields of data-structure inputs. In all the tools presented here, memory configuration and alias relations are handled by an abstract memory-map. The ability to recognise possible alias relations (and cover all paths) relies on the precision of the memory-map and the extent to which alias relations are precisely handled by the constraint solver. Due to their linear constraint solver, DART and CUTE encounter difficulties in the treatment of aliases. In these tools, when constraints with symbolic values cannot be handled by the solver, the symbolic values are replaced by the corresponding concrete values. While this use of concrete values may compensate solver limitations, there is no way to estimate the completeness of these approaches. The constraint solver of EXE uses a dedicated SAT-solver connected to a “bit vectors and arrays” decision procedure (STP)[GD07] in which memory-map and alias relations can be completely represented. PathCrawler’s memory-map is less precise and cannot treat aliases resulting from pointer arithmetic on record types (`struct` in C), type unions or casts between data-structures.

The subject of this paper is the extension of PathCrawler to treat function calls. As in our initial version of PathCrawler, CUTE uses an inlining approach for function calls while this point is not addressed for EXE in [CGP+06]. The only approach which seems to be comparable with our treatment of called functions is a recent extension of DART, named SMART[God07]. We will make a detailed comparison with SMART in Section 5 below but we can already note that our approach uses specifications of called functions while SMART needs their source code. This means that our approach can be used when the source code of called functions is not available (COTS or library components).

We now recall from the proof of PathCrawler’s completeness in [WMM04] the notations that will be necessary in the rest of the paper. Each path predicate PC_i can be given as the ordered conjunction of all successive branch conditions re-expressed in terms of input variables, $C_{i,j}$, encountered along the corresponding path:

$$PC_i = C_{i,1} \wedge \cdots \wedge C_{i,pi} \qquad (1)$$

with pi the number of conditions (such as *while* or *if* instructions) in the path associated to PC_i. As in CUTE and DART, our strategy is a depth-first construction of the tree of feasible execution paths, as shown in Figure 1.

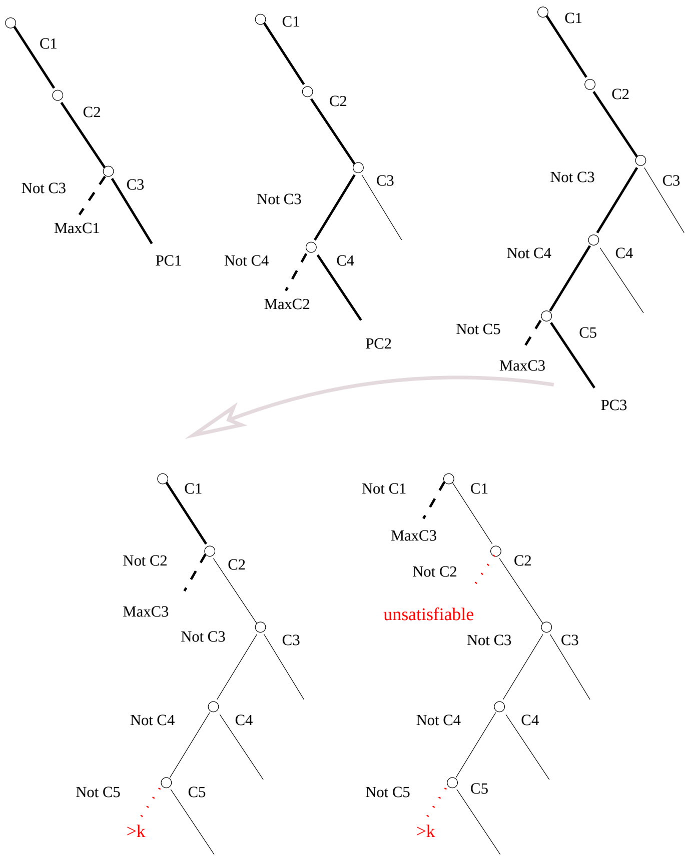

Figure 1. Test selection strategy

To find a solution for the next test-data t_{i+1}, we choose to first solve the longest prefix of PC_i with the last condition negated which we call $MaxC_i$. $MaxC_i$ is defined as:

$$MaxC_i = C_{i,1} \wedge \cdots \wedge C_{i,m-1} \wedge \neg\, C_{i,m} \qquad (2)$$

with $m \in [2..pi]$

If this selected conjunction $MaxC_i$ is unsatisfiable, we try again with the second longest conjunction that is $C_{i,1} \wedge \cdots \wedge C_{i,m-1} \wedge \neg\, C_{i,m-1}$.
Note that when a path predicate prefix has no solution, our strategy does not construct or explore any path predicate extending this prefix because we prune the search tree at the branch associated with this prefix. On the contrary, if $MaxC_i$ is satisfiable, then a new test-data t_{i+1} is selected, corresponding to the path condition PC_{i+1}. The associated path contains at least m conditions but possibly even more corresponding to the complete path whose beginning verifies the first required conditions. Again, we have to consider all the prefixes (which have not previously been explored) of the path corresponding to PC_{i+1} with the last condition negated.

For the treatment of loops and the classical k-path criterion, the definition of $MaxC_i$ is modified to take into account the number of loop iterations (as for $MaxC_3$ in Figure 1). Conditions at the starting point of loops are annotated with the number of loop iterations. By default, when there are loops, the test selection strategy iteratively generates data inputs, with an increasing number of loop executions. If the negation of a condition would result in a loop re-entry after k or more iterations, then it is not explored. When considering the longest conjunction of the current prefix, we never generate a new path predicate prefix containing more than k loop iterations. However, a test-data which is a solution of a such path predicate prefix can occasionally result in the coverage of a path containing more than k iterations of a loop.

The PathCrawler process terminates when there are no more paths to explore.

3 Our treatment of function calls

We choose to abstract the called functions of the function under test by using their specifications [Mou07]. The idea is to abstract the internal structural paths of the called functions by the definition of the corresponding functional domains.

3.1 Specification

The user supplies the specifications of the called functions, validated during a previous test or proof campaign or provided by the authors of off-the-shelf-software. They are expressed in a specification language corresponding to first-order logic on finite domains. We choose to express the specifications of the functions in the form of pre/post couples [Hoa69]. By doing this, we characterise the constraints for the correct use of a function and we identify the sub-domain on the inputs corresponding to each behaviour of the called function.

Every precondition is a set of constraints on entries to be respected for correct behaviour of the function. The precondition, $Pre(g, W)$, of a function g, characterises the function's definition domain, where W is the vector of input variables of the function.

A postcondition is a set of constraints which the function has to respect after execution for a given input sub-domain. We choose the following format of postconditions:

$$Post(g, W, Z) : (D(g, W) \wedge Q(g, W, Z))$$

with $D(g, W)$ an input sub-domain of the function and $Q(g, W, Z)$ characterising the expected behaviour, w.r.t. W and Z for $D(g, W)$, with Z the vector of output variables.

This format is easy for users to understand. Furthermore, it is frequently used to specify conditions in state-transition systems and is already widely used in industry.

By analogy with the Hoare logic triplet [Hoa69], the preconditions $Pre(g,W)$ and every postcondition $Post_i(g,W,Z)$ of g can be represented as :

$$Pre(g,W) \wedge D_i(g,W)\ [g(W)=Z]\ Q_i(g,W,Z)$$

The intuitive sense of the previous Hoare triplet is that if W verifies $Pre(g,W) \wedge D_i(g,W)$ then g ends and then W and Z verify $Q_i(g,W,Z)$.

As in most automatic tools treating Hoare logic, we impose that $Q(g,W,Z)$ is a functional expression of Z according to W i.e. which does not introduce supplementary conditions on W which are not already implied by $D(g,W)$. So the following relation is verified:

$$Dom(Post(g,W,Z))_{|W} = Dom(D(g,W))_{|W}$$

with $Dom(P(W))_{|W}$ representing the domain of the values of W verifying $P(W)$.

We shall note a couple pre/post $PP_i(g,W,Z)$ of a function g :

$$PP_i(g,W,Z) = (Pre(g,W) \wedge D_i(g,W), Q_i(g,W,Z))$$

The specification of the function g, noted $Spec(g,W,Z)$ corresponds to a finite set of pre/post couples for g:

$$Spec(g,W,Z) : \{PP_i(g,W,Z)\}_{i \in I}$$

with I a finite set of numbers.

Example 3.1 *Here is the specification of the called function of Example 1.1,* get_inter*, which returns the interval of an array $w1$ of length $w2$ in which we are looking for the value provided by $w3$.*

```
FUNCTION get_inter
/*@ requires
@ (forall i,(0<=i)&&(w2-1>i)&&(w1[i]<=w1[i+1]))
@*/
/*@ ensures
@ ((w1[0]>w3) => (z1=-2))
@ ((w1[w2-1]<=w3)  => (z1=-1))
@ ((w1[0]<= w3)&&(w1[w2-1]>w3) =>
 (exist i,(0<=i)&&(w2-1>i)&&(w1[i]<=w3)&&
(w3<=w1[i+1]))&&(z1=i))
@*/
END
```

The array $w1$ is sorted in increasing order, the function returns -2 if the sought value is lower than the first element of the array and -1 if the value is greater than the last element. The concrete syntax used below is the one of our PathCrawler tool. The different elements involved can be easily understood. The function get_inter *is defined for all values and is specified by means of three pre/post triplets. For example, the first one is defined by* $D_1(\texttt{get\_inter},(w1,w2,w3)) = (w1[0] > w3)$ *and* $Q_1(\texttt{get\_inter},(w1,w2,w3),z1) = (z1 = -2)$.

The implementation of our treatment of function calls imposes certain limitations on the specifications supplied by the user.

Firstly, we need specifications which are complete for all the contexts of called functions: the domains of the call sites for the called function must be covered by the specification. Secondly, we need deterministic specifications : for any instantiation of the input vector W satisfying the precondition, there exists at most one instantiation of the output vector Z such that the postcondition is satisfied (note that for the last pre/post couple in Example 3.1, the specification is deterministic in spite of the presence of the existential quantifier).

3.2 Abstract graph

As for the representation in the form of CFG of the code source of a function, we represent the specification of a called function g in the form of a graph. We use a representation close to the CFG of a function i.e. a connected and directed graph with a unique entry and a unique exit. The conditions associated with the arcs correspond to the conditions on the entries of the pre/post couples of the called function $Pre(g,W) \wedge D_i(g,W)$ (with the hypothesis that the function is called only on its definition domain). Nodes represent meta-instructions $FIND(Z|Q_i(g,W,Z))$ determining an instantiation for Z with the input/output relation defined by the conditions of $Q_i(g,W,Z)$. As the function specification is complete, at least one pre/post couple is activated for any call if the precondition is respected. There is thus at least one feasible path in the abstract graph of a called function for each of its calls. Finally, as the specification is deterministic, the meta-instructions $FIND(Z|Q_i(g,W,Z))$ determine a unique instantiation of Z verifying $Q_i(g,W,Z)$ for a given instantiation of W.

Definition 3.1 *For a function* g *defined by its specification $Spec(g,W,Z)$ with n pre/post couples, the associated abstract graph is a graph with $2+n$ nodes with E the unique entry node, S the unique exit node and n nodes n_i labelled by the meta-instructions $FIND(Z|Q_i(g,W,Z))$ built from the specification.*

The abstract graph of the function g *has $n*2$ edges, including n edges (E,n_i) labelled by the conditions on W $Pre(g,W) \wedge D_i(g,W)$.*

The semantics of an abstract graph such as the graph shown in Figure 2 corresponds to the semantics of a conditional structure (switch).

Example 3.2 *The abstract graph of the called function* get_inter*, used in the previous examples 1.1 and 3.1, is shown in Figure 2. We add a node to the beginning and the end of the abstract graph called $Eget\_inter$ and*

$Sget\_inter$ to link it with the structural graph of the calling function.

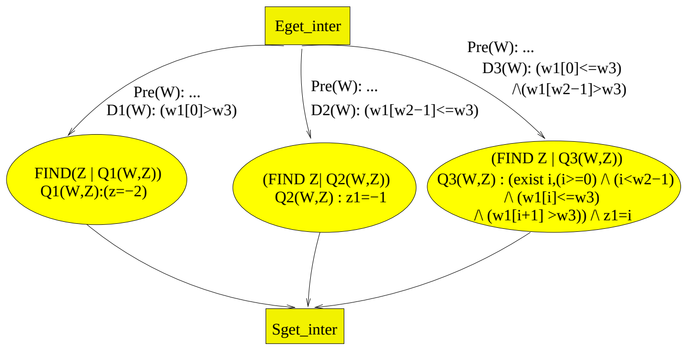

Figure 2. Abstract graph of `get_inter`

3.3 Mixed graph

To build the mixed graph of a calling function, abstract graphs of called functions replace their CFG during the unfolding of the CFG of the calling function. First of all, the source code of the calling function is modified such that calls are isolated while preserving the semantics of the calling function. So, we ensure that each node of the CFG contains at most one call instruction. In fact, the source code of the function under test undergoes an initial pretreatment using the static analysis tool CIL [Lan06]. One of the immediate consequences is that an instruction possesses at most one expression with side effects. So a call instruction cannot be used to label an edge. Every node containing a call instruction in the CFG of the function under test is replaced by the abstract graph of the called function.

We thus obtain the mixed graph of the function under test containing the semantics of the C language for the function under test and the semantics of the specification for the called functions.

The insertion of an abstract graph consists of three steps. The first one allocates to the abstract input parameters of a called function the values of the actual call parameters expressed in terms of the concrete variables. After determination of the abstract outputs, the second step allocates, to a subset of the concrete variables of the function under test the values of the abstract output variables. The third step inserts the FIND meta-instructions into the mixed graph.

Example 3.3 *The mixed graph of our example is shown in Figure 3. For each function call, the two nodes $Eget\_inter$ and $Sget\_inter$ of the abstract graph shown in Example 3.2 are labelled with the mapping between the abstract variable names appearing in the specification and the concrete C variable names or access paths used in the source code of the function under test .*

4 Path coverage of mixed graph

In the simple unit-testing method PathCrawler[WMMR05], the C instructions are translated into Prolog clauses during the pretreatment of the functions. Now, the functional description of the called graph is also translated into Prolog clauses. These are interpreted as constraints at the time of the activation of the execution path and at the construction of the associated path predicate.

In the same way as the structural constraints encoded in the instrumentation of the source code, the constraints encoded from the abstract graph of a called function allow us to define the path predicate as well as the selection domain of the next test-case.

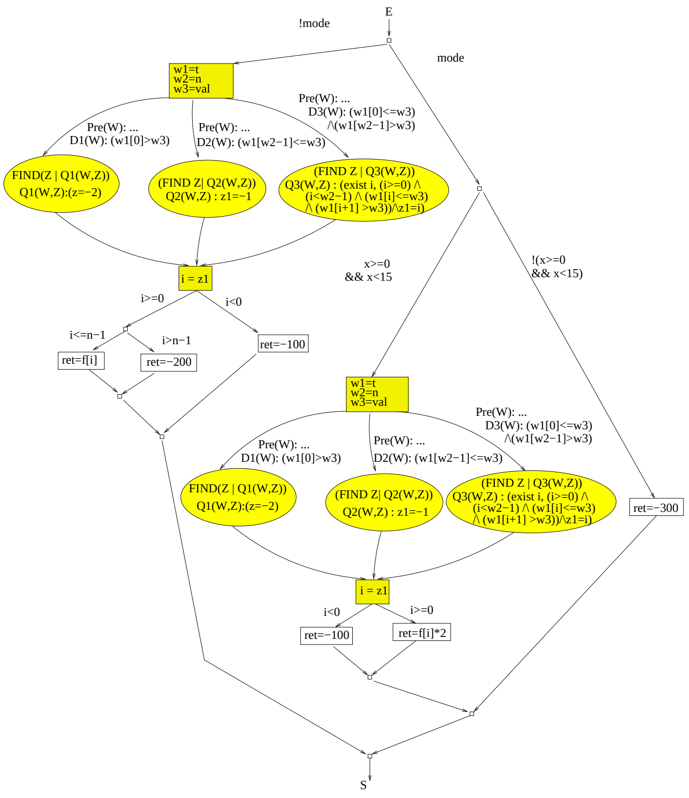

Figure 3. Mixed graph of `apply_f`

The encoding of the meta-instruction $FIND$ is direct in our context. Indeed, if one of the branches of our abstract graph was exercised, the associated constraints $Pre(g, W) \wedge D_i(g, W)$ are verified. The submission of these constraints and of $Q_i(g, W, Z)$ in our constraint solver corresponds to a CSP[1] whose solution is a valid instantiation of Z. The completeness on the calling contexts and the

[1]Constraint Satisfaction Problem

determinism of the specifications guarantee the existence of a unique instantiation of Z respecting the CSP for a given instantiation of W.

4.1 AMP strategy

The first proposed strategy is the most natural: it corresponds to the coverage of all the feasible paths of the mixed graph, i.e. all feasible paths of the function under test and all the functional domains for every calling context of the called functions. We shall call the associated criterion "all-mixed-paths" (AMP).

The application of this criterion is immediate here: we submit the mixed graph to the standard selection strategy of PathCrawler. The mixed graph is then totally covered by a depth-first exploration.

However, if we only need to cover all the paths in the calling function, then the AMP criterion sometimes results in redundant tests. These are the tests which cover mixed paths differing only in their abstract parts issued from specifications of called functions. They exercise different functional domains within a called function but are identical within the calling function.

4.2 MCMG strategy

To treat all-paths coverage of the calling function only, we introduce our second criterion, called "minimal-coverage-of-mixed graph" (MCMG). This requires 100% coverage of all feasible paths of the calling function, without imposing any particular coverage of the called function.

The application of this new criterion requires a modification of PathCrawler's test selection. Where the strategy would usually explore a new functional domain of a called function, two scenarios are now possible. The choice depends on whether all path suffixes after the return from the called function are already covered for the context of the current call. If this is the case, then we do not need to explore any more functional domains in this calling context. Instead, we go back up the path predicate to negate a condition preceding the function call, as for the prefix $MaxC_2$ in Figure 4. Otherwise, we explore a new functional domain in the called function but limit the exploration of the path suffixes after return from the calling function to the suffixes which have not yet been covered in this calling context (as for $MaxC_3$ in Figure 4).

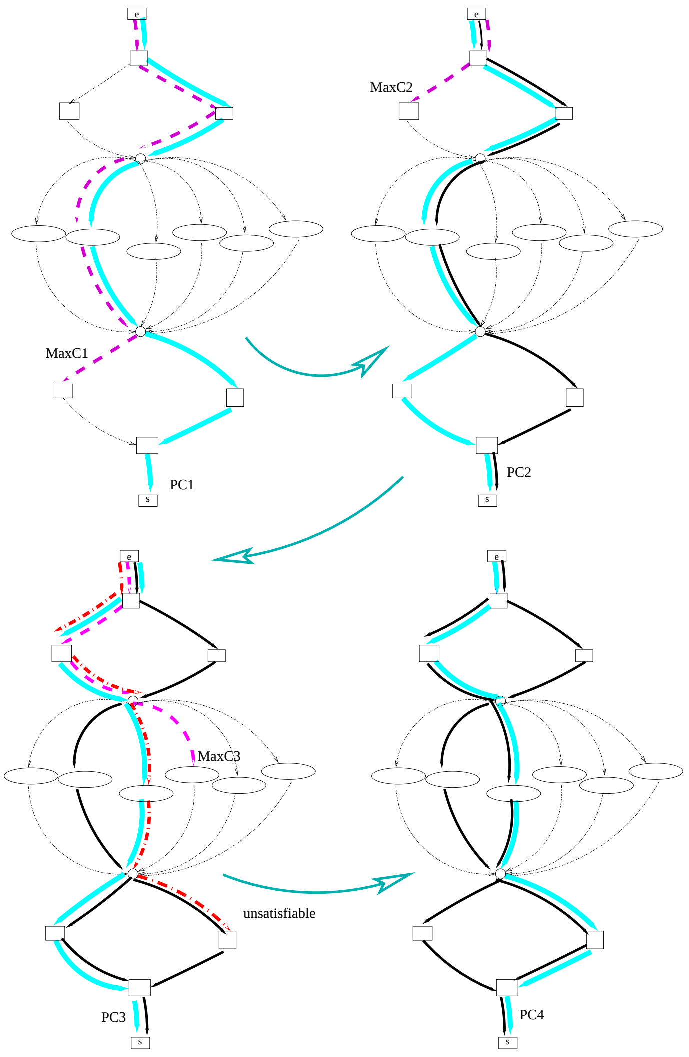

Figure 4. Second strategy: criterion MCMG

5 Discussion and Results

In this section, we will not present new benchmarks but will illustrate the advantages of our approach both on our motivating example and on other examples taken from [God07]. This paper describes how SMART replaces function calls by function summaries. These are logical disjunctions of path predicates (together with the path calculation). DART[GKS05] is run on the calling function and when a call of a particular function is encountered for the first time then a function summary is built. This is done by running DART on the called function in this calling context and forming the disjunction of the path predicates and calculations for every path covered. At the next call of the same function, if the concrete input values of the called function respect one of the path predicates in the summary, then the function call is replaced by this summary. If not, a new function summary is built for this calling context. Thus, at the next encountered function call, the method will compare the new concrete input values with each of the function summaries.

This approach avoids repeating the exploration of the same called function when it is called several times with compatible calling contexts. However, it imposes the manipulation of potentially enormous disjunctions of constraints which risk being intractable (even for modern SAT solvers). Furthermore, according to our understanding of [God07], by testing for intersection of calling contexts rather than inclusion, SMART runs the risk of incomplete coverage of the calling function, as shown on our example in the next Section.

5.1 Our motivating example

Let us recall our motivating example, whose code was presented at the beginning of this paper in Example 1.1 and its specification in Example 3.1. The calling function `apply_f` contains 6 different execution paths, 5 of which contain a call to the function `get_inter`. Note that only 4 paths are feasible: the two paths evaluating the condition at lines 30 and 40 as true are infeasible.

The called function `get_inter` contains 10 feasible paths: 2 paths outside the `for` loop and 8 paths in the loop. We thus have 31 possible execution paths if the called function is inlined, of which only 16 are feasible. We will not enumerate these test-cases but they correspond to the possible combination of feasible structural paths of the calling function and feasible structural paths of the called function.

If we observe the specification of the called function, there are only 3 functional domains. The corresponding mixed graph, shown in Figure 3, contains 16 different mixed paths (not the same as the 16 feasible paths above). We have already noted the presence of dead code in Example 1.1, it means there are infeasible mixed paths.

With the AMP strategy, we obtain 5 test cases. Figure 5 shows the simplified mixed graph of `apply_f` (cf. Figure 3) and the mixed paths which are covered.

In Figure 5, we can observe that only one of the two paths 4 and 5 is needed to obtain the coverage of all structural paths of the calling function. Indeed, with our second strategy MCMG, we obtain only 4 test cases, covering only paths 1 to 4 of Figure 5.

In this previous example, the called function is called in two different contexts. If the first calling context to be explored is the one with the more restricted domain (at line 29 in Example 1.1), then SMART would use the summary of this first context for the exploration of the second one (at line 36). But as the first calling context is more restricted, two paths in the called function which are feasible in the second calling context will not be represented in the function summary. These are the paths which have true conditions at line 8 and line 11. These paths in the called function will therefore not be explored in the second calling context. But one of these paths must be taken in the called function

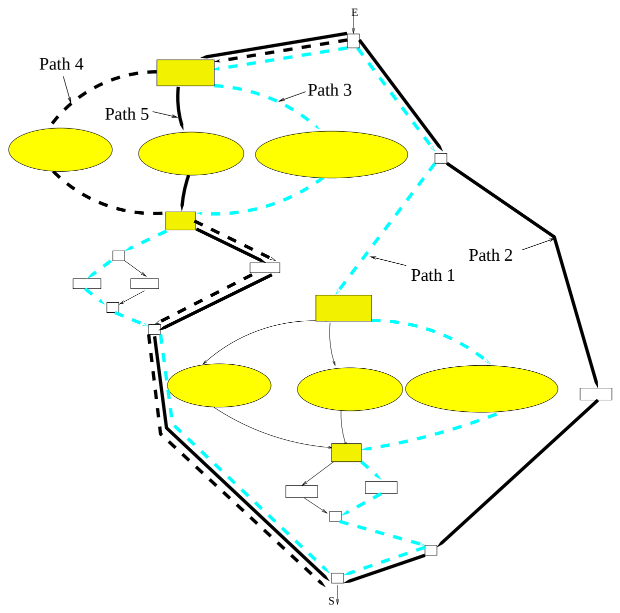

Figure 5. Mixed paths covered for `apply_f`

in order to follow the path suffix which has a true condition at line 34. In this case, SMART would not cover paths with this suffix. This is why SMART cannot always cover as many paths in the calling function as those covered by DART with function inlining.

5.2 Other examples

5.2.1 First example from [God07]

We use one of the examples applied to the method SMART in [God07]. The code of the function under test is:

```
int err;
int is_positive(int x)
{
 if (x>0)
  return 1;
 return 0;
}
void top (int s[N]){
 int i, cnt=0;
 err=0;
 for(i=0;i<N;i++)
  cnt=cnt+is_positive(s[i]);
 if(cnt==3)
  err=1;
 return;
}
```

We can see that only two paths of the function under test `top` are feasible while the inlining treatment of the called function `is_positive` covers 2^N paths. Following an inlining treatment, we have covered this function with $N = 10$ and we obtain 1024 test cases in 2.27s cpu on Linux 2GHZ. This number of feasible paths can be explained by 2 facts: (1) two paths of the function under test correspond to the paths verifying or not the conditional instruction `if(cnt==3)`, (2) the loop `for` contains only one feasible path because this loop has a fixed number of iterations.

SMART performs only 4 runs to cover the function under test : 2 runs to build the path summaries of the called function `is_positive` and 2 runs using these summaries to cover the 2 paths of the calling function.

If we apply our AMP strategy to the corresponding mixed graph, we cover all 2^N paths of the mixed graph. With $N = 10$, we obtain 1024 test cases in 2.21s cpu on Linux 2GHZ. Our AMP strategy has a similar execution time to that of function inlining. This is easy to explain: the called function contains 2 structural paths and 2 pre/post couples in its specification. The associated specification of the called function is:

```
FUNCTION is_positive
/*@ requires
@ (true)
@*/
/*@ ensures
@ ((w1>w2) => (z1=1))
@ ((w1<=w2) => (z1=0))
@*/
END
```

We can see that this specification is similar to the implementation and provides no combinatorial benefit. We see here that our AMP strategy is only effective if the called function possesses more feasible structural paths than functional domains.

However, if we apply the second strategy, MCMG, we cover just the 2 test-cases corresponding to the 2 feasible paths of the function under test. With $N = 10$, we cover 2 test cases in 0.09s cpu on Linux 2GHZ. Obviously, the strategy MCMG is more effective for this example.

5.2.2 Second example from [God07]

Now, let us take another example extracted from[God07]. The called function is:

```
//locate index of first character c
// in null-terminated string s
int locate(char* s,int c){
int i=0;
while(s[i]!=c)
{ if (s[i]==0)
  return -1;
i++}
return i;
}
```

The inputs are a string `s` of maximum size n with `s[n]` zero and a character `c` to be searched for in this string. The called function `locate` contains $2 * n$ execution paths if `c` is nonzero and n paths if `c` is zero. The function under test is:

```
1  void top(char *input){// assume input is null-terminated
2   int z;
3   z=locate(input,'a');
4   if (z==-1) return -1;
5   if (input[z+1]!=':') return 1;
6   return 0;
7  }
```

The calling function `top` contains 3 possible execution paths. With an inlining method, there are $3 * n - 1$ feasible paths.

SMART systematically executes all possible paths of these functions separately. For the called function `locate`, SMART builds a summary which is the disjunction of $2 * n$ terms for the $2 * n$ paths. Next, SMART tries to explore the 3 paths of the function under test `top` using the summary to replace the called function. For this example, SMART executes $2n + 3$ runs so the number of runs grows in a linear way with the size of the input string.

Using our AMP strategy, the called function is replaced by the corresponding abstract graph which contains only 2 abstract paths: the called function possesses 2 functional domains (the string does or does not contain the character). With this strategy, we would execute only 3 feasible mixed paths. One path leads to the execution of the return instruction at line 4 of the calling function `top`. Another path leads to the return instruction at line 5 of the function and the last path leads to the return at line 6. For each of these paths, only one functional domain can be activated. The 3 mixed paths covered by the AMP strategy correspond exactly to the 3 feasible paths of the function under test so the MCMG strategy executes the same test cases.

In our approach, the number of cases to be considered depends on the number of pre/post cases in the specification instead of the number of feasible paths of the called functions. If there are very many paths then with a declarative specification the user can often limit the complexity of test-case generation. This can be seen as an advantage of our approach. The other advantage being that it preserves the completeness of the coverage of paths in the calling functions.

6 Conclusion

Benchmarking on a range of examples is still in progress. However, the experiments described in this paper showed decisive results. The exploration of the mixed graph according to the AMP criterion shows that our objectives are reached: the coverage of the function under test is maintained and the exploration of the called functions is limited[2]. The application of the MCMG criterion on the mixed graph allows the exploration of the called functions to be limited even further by eliminating the test-cases which are not needed for the coverage of the calling function alone.

We have seen an example of grey-box testing as the joint use of specification and code source for test generation. This technique could be put to other uses.

[2]According to the reasonable hypothesis that there are more structural paths than functional domains

For example, we can envisage the resolution of the "missing path" problem. A missing path is a functionality of the specification which "has not been implemented"[GG75]. In fact, as soon as we dispose of a precise specification, this problem is addressed by conformance testing. In our case, the idea would be to apply the PathCrawler strategy on each input subdomain defined in the specification (i.e. from each $Dom(Pre(g,W) \wedge D_i(g,W)))$. This would ensure a structural coverage of every identified functional subdomain. The corresponding postconditions would be used to set up an automatic oracle during PathCrawler exploration.

The AMP strategy covers all the functional domains of the called functions for each of their call contexts. This is why it could also be used instead of unit testing of the called functions. This point would allow us to dispense with our basic hypothesis that called functions are already tested. Indeed, the AMP strategy allows the implementation of a technique of in-context testing for the called functions. An in-context oracle can be built from a comparison between the outputs obtained at each execution and those calculated by the abstract graph.

Finally, assuming we do not dispose of a convenient specification for called functions, the exploration of their source code could be limited with a strategy inspired from the MCMG strategy. The idea is to cover only those paths in the called functions which are necessary for the complete coverage of the paths of the calling function.

References

[BM93] W. G. Bently and E. F. Miller. Ct coverage–initial results. *Software Quality*, 2(1):29–47, 1993.

[CGP+06] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. In *ACM Conf. on Computer and Communications Security*, pp. 322–335, 2006.

[FK96] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Trans. Soft. Eng. Methodol.*, 5(1):63–86, 1996.

[GBR00] A. Gotlieb, B. Botella, and M. Rueher. A CLP framework for computing structural test data. *LNCS*, 1861:399–413, July 2000.

[GD07] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV*, pp. 519–531, 2007.

[GG75] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Trans. Soft. Eng.*, 1(2):156–173, 1975.

[GKS05] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *ACM SIGPLAN 2005 Conf. PLDI'05*, pp. 213–223, 2005.

[GMS99] N. Gupta, A. P. Mathur, and M. L. Soffa. UNA based iterative test data generation and its evaluation. In *Proc. ASE'99*, pp. 224–232, 1999.

[GN97] M. J. Gallagher and V. Lakshmi Narasimhan. Adtest: A test data generation suite for ada software systems. *IEEE Trans. Soft. Eng.*, 23(8):473–484, 1997.

[God07] P. Godefroid. Compositional dynamic test generation. In *Proc. POPL '07*, pp. 47–54, 2007.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Com. of the ACM*, 12(10):pp. 567–580, 1969.

[Kor96] Bogdan Korel. Automated test data generation for programs with procedures. In *Proc. ISSTA'96*, pp. 209–215, 1996.

[Lan06] CIL : C Intermediate Language. *CIL - Infrastructure for C Program Analysis and Transformation*. 2005/2006. http://manju.cs.berkeley.edu/cil/.

[MA00] B. Marre and A. Arnould. Test sequences generation from Lustre descriptions : GATeL. In *Proc. ASE'00*, pp. 229–237, 2000.

[MM98] C. C. Michael and G. McGraw. Automated software test data generation for complex programs. In *Proc ASE'98*, pp. 136–146, 1998.

[Mou07] P. Mouy. *Automatisation du test de tous-les-chemins en présence d'appels de fonction*. PhD thesis, Université d'Evry Val d'Essonne, 2007.

[PM87] R. E. Prather and J. P. Myers, Jr. The path prefix software testing strategy. *IEEE Trans. on Soft. Eng.*, 13(7):761–766, 1987.

[SD01] N. Tran Sy and Y. Deville. Automatic test data generation for programs with integer and float variables. In *Proc. ASE'01*, pp. 13–21, 2001.

[SMA05] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *Proc. ESEC/FSE'05*, pp. 263–272, 2005.

[WMM03] N. Williams, B. Marre, and P. Mouy. On-the-fly generation of structural tests for C functions. In *Proc. ICSSEA'03*, 2003.

[WMM04] N. Williams, B. Marre, and P. Mouy. On-the-fly generation of k-paths tests for C functions : towards the automation of grey-box testing. In *Proc. ASE'04*, pp. 290–293, 2004.

[WMMR05] N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler : Automatic generation of path tests by combining static and dynamic analysis. In *Proc. EDCC'05*, pp. 281–292, 2005.

[WNJ97] M. Wallace, S. Novello, and J.Schimpf. *ECLiPSe: A platform for Constraint Logic Programming*. IC-Parc, Imperial College, London, 1997.