On-the-Fly Generation of *K*-Path Tests for C Functions : towards the Automation of Grey-Box Testing

Nicky Williams, Bruno Marre and Patricia Mouy
CEA/Saclay, DRT/LIST/ SOL/LSL,91191Gif sur Yvette, France
{Nicky.Williams, Bruno.Marre, Patricia.Mouy}@cea.fr

Abstract

We present a method for the automatic generation of tests satisfying the all-paths criterion, with a user-defined limit, k, *on the number of loop iterations in the covered paths. We have implemented a prototype for C code. We illustrate our approach on a representative example of a C function containing data-structures of variable dimensions, loops with variable numbers of iterations and many infeasible paths.*
We explain why our method is efficient enough to scale up to the unit testing of realistic programs. It is flexible enough to take into account certain specifications of the tested code. This is why we believe that it could become the cornerstone of a fully automatic grey-box testing process, based on the novel combination of code instrumentation and constraint solving described here.

1. Introduction

Rigorous testing of delivered software, by its implementers or by external certifiers, is increasingly demanded, along with some quantification of the degree of confidence in the software implied by the test results. The reasons for this include the increase in the deployment of embedded software systems and the re-use of off-the-shelf components. This sort of testing cannot be based on a restricted set of hand-crafted test objectives or use-cases, which may have to be manually updated if the software requirements change. Testing must be made as automatic as possible, with automatic generation of a large number of test cases according to a well-justified selection criterion.

Our ultimate aim is the implementation of a fully-automatic test process based on a grey-box test selection strategy. This will advantageously combine white box (structural) and black-box (functional) strategies in the following way: for each specified functionality of the program under test, the set of input values which will demonstrate this functionality will be calculated and tests will be generated for full structural coverage of this reduced set of input values. The specification of the functionality will be used to automatically generate an oracle for these structural tests.

A fundamental step in the implementation of this testing system is an efficient technique for the automatic generation of test sets for structural coverage. There has been much research on the automatic generation of structural test cases but many techniques do not scale up to full coverage of realistic-sized programs. We believe that this is because they were not really designed for full coverage. This article describes a novel method for the efficient generation of tests for 100% coverage of feasible execution paths. We show how it can easily be modified to satisfy the well-known *k*-path limitation on the number of loop iterations covered. Moreover, we show the beginnings of the integration into the grey-box strategy by taking into account a precondition on the inputs to the program under test.

There has been much debate about the relative merits of structural and functional testing and most researchers now accept that both are necessary. Structural testing has the following disadvantages:

- It is often based on line- or branch-coverage criteria whose justification remains weak. Indeed, the (often implicit) justification for both white box and black box testing is a uniformity hypothesis [3]: each tested input vector is a representative of an equivalence class of other input vectors and success of the test for this input vector implies that the test would also have succeed for all other input vectors in the equivalence class. The justification for this hypothesis seems strongest in the case of coverage of all feasible execution paths: the equivalence class is all inputs which activate exactly the same straight-line calculation.
- The link with the requirements remains informal: often the user must manually check large numbers of test results. There is also the "missing path" problem: thoroughly exercising the implementation tests the functionalities it does fulfil but does not demonstrate that it fulfils all the required

functionalities. These problems can be resolved by the grey-box strategy outlined above.

- It requires large numbers of tests. Unfortunately, this is the price to pay for a well-justified automatic selection strategy. Structural coverage must be just a part of a carefully designed overall system test strategy. Above all, as much of the test process as possible must be automated, including the evaluation of the test results. In the example described in this paper, a hand-coded oracle was inserted into the automatic testing loop but ideally oracles would be automatically generated from specifications of the program under test, or assertions in the code. Current research [12][13] leads us to believe that this is a feasible prospect.

2. Related Work

Most previous work on test data generation for structural testing of sequential programs addresses the problem (called the Test Data Generation Problem (TDGP) in [11]) of finding data to cover a test objective in the form of given node, branch or path of the control flow graph.

Static approaches to test case generation [6][8][19] typically extract the constraints on input values (path predicate) corresponding to a path from the control flow graph covering the test objective and then solve these constraints to find a test case which activates the path. In theory, symbolic execution can be used to construct the path predicate. However, in practice symbolic execution encounters problems in the detection of infeasible paths (notably in the case of loops with a variable number of iterations), the treatment of aliases and the complexity of the formulae which are gradually built up. Various ways around these shortcomings have therefore been proposed. In [6], the source code is first transformed into SSA form and then constraints (or, in the case of loops, constraint generators) are constructed and submitted to a constraint solver throughout symbolic traversal of the desired path in the control flow graph. This avoids complicated formulae and should help infeasible paths to be detected as soon as possible but does not alleviate the problem of the treatment of aliases. Instead of symbolic execution and constraint solving, [17] uses an analysis which closely resembles the abstract interpretation used in [7]. Abstract interpretation constructs an approximate set of solutions which may be larger [7] or smaller [17] than the true set.

Note that solving non-linear constraints is decidable only for data types with finite domains (integers, booleans and other enumerated data types). However, fixed- and floating-point numbers can also be treated as numbers with finite domains (instead of considering them as real numbers and then trying to take rounding errors into account). Specialised techniques to correctly handle arithmetic operations on floating-point numbers are currently being investigated in research [16][19] which holds the promise of decidable and precise constraint solving for these numbers too. Solving, or determining the satisfiability of, a set of linear or non-linear constraints on numbers with finite domains is NP-complete in the worst case but heuristics have been developed which seem to give an acceptable performance for most test case generation problems [8][14].

Dynamic approaches [5][11][15] avoid the problems of symbolic execution by running an instrumented version of the program under test on arbitrary data and then using function minimisation to try to modify the data so that a path covering the test objective is followed. The function to minimise is based on an evaluation of the “distance” from satisfaction (based on the difference between the result of the branch condition and the desired result) of branch conditions of the executed path. The program is instrumented so as to indicate the branches taken and evaluate the distance from satisfaction at the first branch which deviates from the desired path (or paths). The path predicate is not needed because general heuristic function minimisation techniques are used to search for input values to reduce to zero the simple number representing “distance” from the desired path. The disadvantages of these techniques are that they may need a great many executions before a test case is found, they may fail to find a test case even when one exists and they do not terminate if the desired path is actually infeasible.

[9] differs from other dynamic approaches in not using heuristic function optimisation but, as in static approaches, solving constraints derived from the predicate of a path covering the test objective. However, instead of using symbolic execution to extract the path predicate, program fragments obtained by a form of static slicing are repeatedly executed on different inputs. This gives a linear approximation in the case of non-linear branch predicates. Further executions of other program slices can then be used to measure the difference between the result of each branch condition and the desired result and derive a set of linear constraints on the increments to the arbitrary input. Rather than using a generic constraint solving tool, this last stage is repeated in order to solve the constraints using an iterative technique which initially treats all variables as real numbers. Extra iterations are needed to move from a solution using real numbers to one for integer inputs. When the underlying branch conditions are not linear, the technique may fail to converge towards a solution and it must be supposed that the path is infeasible.

[18] predates all the previously mentioned work and describes an algorithm rather than a prototype implementation. However, it is based on the observation that if full structural coverage is needed, then instead of deciding in advance the order in which tests will be

generated, test selection can be based on the path predicate of the previously generated test case, and indeed on a prefix of this predicate.

The contributions of this paper are the following:

1) We propose a new method for generation of all-paths and *k*-paths test input data
2) We maintain that the TDGP is not the best formulation of the problem of generation of a test set for full structural coverage. We do not need to construct the control flow graph. Instead, we iteratively cover "on the fly" the whole input space of the program under test. This is similar to the idea originally proposed in [18], but we apply it to path coverage instead of branch coverage and we do not risk leaving feasible paths uncovered by limiting exploration of each previous path predicate to only one prefix. In the TDGP, the next path, or paths, to be covered must be selected from the control flow graph. This poses the problem of the elimination of paths which exist in the graph but are infeasible in reality, notably paths containing loops with a number of iterations that depends on input values. This problem is reduced in our approach to the detection of the infeasibility of negating the last condition in a satisfiable path predicate prefix. Infeasibility is detected as soon as the shortest infeasible prefix has been constructed and we can immediately eliminate, without even constructing them, all the paths containing this prefix. We thus avoid the combinatorial explosion of infeasible paths encountered by other approaches and which prevents them from being scaled up to path coverage of realistic programs.
3) Like the dynamic approaches to test data generation, our method is based on dynamic analysis, but instead of heuristic function minimisation, we use constraint logic programming to solve a (partial) path predicate and find the next test case, as in the approaches based on static analysis. We suffer neither from the approximations and complexity of static analysis, nor from the number of executions demanded by the heuristic algorithms used in function minimisation and the possibility that they fail to find a solution.

Our approach is applicable to all imperative languages and is currently being implemented for C. Throughout this paper, we will use as an example the C function Merge, whose source code is shown in Figure 1. This function takes as input two arrays, `t1` and `t2`, of ordered integers and the effective lengths, `l1` and `l2`, of the arrays and outputs in array `t3` the ordered elements of `t1` and `t2`. In the following sections, we give an overview of the approach and then describe in more detail its principal stages: Instrumentation, Substitution and Constraint Solving. We then describe the results of applying our prototype implementation to the function Merge, before concluding with a discussion of further work.

```
void Merge (int t1[], int t2[],
            int t3[], int l1, int l2){
  int i = 0;   int j = 0;   int k = 0;
  while (i < l1 && j < l2) {
    if (t1[i] < t2[j]) {
      t3[k] = t1[i];
       i++; }
    else {
      t3[k] = t2[j];
       j++; }
    k++; }
  while (i < l1) {
   t3[k] = t1[i];
    i++;
    k++; }
  while (j < l2) {
   t3[k] = t2[j];
    j++;
    k++; }
}
```

Figure 1 : source code of the function Merge

3. Our approach

Our approach (see Figure 2) starts with the instrumentation of the source code so as to recover the symbolic execution path each time that the program under test is executed. The instrumented code is executed for the first time using a "test case" which can be any set of inputs from the domain of legitimate values. The symbolic path which we recover is transformed into a path predicate by substituting the assignments into the conditions so as to obtain a conjunction of constraints on the values of the input variables. This path predicate defines the "domain" of the path covered by the first test case, i.e. the set of input values which cause the same path to be followed. The next test case is found by solving the constraints defining the legitimate input values outside the domain of the path which is already covered. The instrumented code is then executed on this test case and so on, until all the feasible paths have been covered.

The strict interpretation of the all-paths criterion soon becomes unrealistic for programs containing loops with a variable number of iterations, such as the three loops in our example function Merge. Generation of one test for each possible number of iterations of each loop in each path results in a combinatorial explosion in the number of tests. The all-paths criterion is therefore often relaxed to impose coverage of only those paths containing numbers of iterations within a user-defined limit, *k*. In our example, *k* is set to 2 so only the feasible paths containing combinations of 0, 1 or 2 loop iterations are covered.

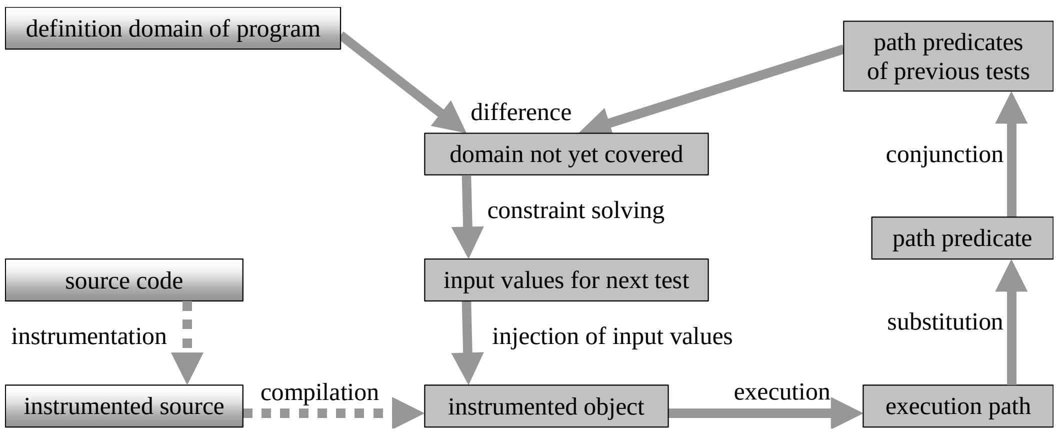

Figure 2 : our approach

In order to implement this *k*-path criterion, we have extended the instrumentation of the source code so as to annotate the conditions which determine loop entry or exit. Our constraint solving strategy uses these annotations.

4. Instrumentation

The instrumentation stage is an automatic transformation of the source code so as to print out the execution path, i.e. each assignment carried out and each condition satisfied during execution. A trace instruction is therefore inserted after each assignment and each branch of the source code. The instrumentation is based on an initial lexical and syntactic analysis, followed by an interpretation of the C code. We implemented it by modifying and re-using two modules from an existing static analysis tool (Caveat [1], developed by the CEA).

The first module transforms the code so as to decompose certain statements and eliminate

- multiple assignments in the same statement
- multiple function calls in the same statement
- conditional statements with side effects and
- composite conditions in conditional statements.

Note that the decomposition of multiple conditions reinforces the test criterion which really lies somewhere between all-paths and all-paths combined with the branch criterion MC/DC of the avionics software development guideline DO-178B.

The second module is an interpreter of the transformed code which enables the insertion of the trace instructions. Along with pointers, the C language offers alternative notations to access elements of structured data (arrays, structures or unions) but in our trace instructions, all data access paths are represented in a canonical form. This rewriting of access paths, which is purely syntactic, simplifies the substitution stage.

In fact, the execution path as output by the trace instructions constitutes a semantics of the executed statements. In order to indicate to the user the path which is covered by the test, the trace instructions also add an annotation indicating the arc followed in the control flow.

To implement the *k*-path criterion, the annotation of conditions in the head of loops also includes the following information : which loop this is the head of, whether the condition corresponds to loop entry or exit and whether it, or its negation, ensures definitive re-entry into the loop. For example, the trace of the failure of the first test in Merge, `i < l1,` codes the following information:

- the instruction is of type condition (and not assignment)
- the identification of the condition in the code (negation of 1st (sub-)condition in source code)
- this condition results in exit from loop number 1
- its negation (success of the test `i < l1`) does not necessarily ensure definitive re-entry into the loop (this is because it is not the last sub-condition of the conjunction).

5. Substitution

A path predicate is a conjunction of constraints expressed in terms of the values (at input) of the input variables. However, the symbolic conditions output by the instrumentation of the conditional statements in the source code may be expressed in terms of local variables (or intermediate values of input variables). The substitution stage of our approach carries out the projection of these conditions onto the values of the inputs. The sequence of statements output by the execution of the instrumented program is traversed and each assignment is used to update a “memory map” which stores the current symbolic value of local variables in terms of the input values. When a condition is encountered, all occurrences of local variables are replaced by their current symbolic values. The resulting list of conditions is the path predicate. Because we analyse a single, unrolled, path, we do not need to use the SSA form used in [6]. The size of our memory map could be reduced by a static analysis of the source code before the instrumentation which, without considering control flow, could identify any output variables whose values are never referenced elsewhere.

The presence of aliases (two or more ways of denoting the same memory location) in the code complicates the updating of the memory map and the comparison of conditions and assignments, but the problem posed by aliases in our approach is simpler than that posed in more general static analysis contexts. This is because we know the symbolic value of any local variable appearing in a C access path. It is only when an access path contains either an input variable or a local variable whose value depends on that of an input variable that we encounter a potential alias problem. In this case, the effect of some assignment on a subsequent condition could depend on the satisfaction of a certain relation on the values of the inputs. Sometimes, several different relations on input values could lead to the same result of the conditional statement. In other words, the path predicate could contain a disjunction in which each term contains a different “alias” relation on the input values, i.e. a relation which is not directly induced by a condition in the path but which leads to a certain interpretation of the path conditions.

```
void f (int x, int y, int tab[]){
  int v;
  int *pt;
  v = x * 2;
  v = v - y;
  pt = &tab[2];
  tab[2] = x;
  *(pt + v) = y;
     if (tab[y + 4] > 5)
  ...
```

Figure 3 source code of alias example

For example, the condition at the end of the code fragment shown in Figure 3 (which we will use to illustrate our treatment of aliases because the function Merge does not contain any access paths depending on input values), references un unknown element of the array `tab` in which the value of one element has been set to the input value *x*, one (possibly the same) to the input value *y* and the other elements still have their value at entry to the function. The path predicate for the code fragment is:

(	$y + 4 = 2 + x*2 – y$	(alias relation)
$\wedge$	$y =< 5$)	(failure 1st condition)
$\vee$ (	$y + 4 \neq 2 + x*2 - y$	(alias relation)
$\wedge$	$y + 4 = 2$	(alias relation)
$\wedge$	$x =< 5$)	(failure 1st condition)
$\vee$ (	$y + 4 \neq 2 + x*2 - y$	(alias relation)
$\wedge$	$y + 4 \neq 2$	(alias relation)
$\wedge$	$tab[(y + 4)] =< 5$)	(failure 1st condition).

In such cases, we can find the alias relation effectively satisfied by the previous test case, by referring to the actual input values. Our strategy is to insert just this alias relation, which we know to be feasible, into the “path predicate”. Because our method then treats the same path with a different alias relation as an uncovered path, and looks for a test for it, the effect is to eliminate disjunctions in the path predicate by creating one test for each feasible alias relation of the same path. In other words, any path predicates containing disjunctions are decomposed into a set of (consistent) conjunctions, each representing a different “path predicate” (because the alias relation is different) and being covered by a different test, even though they all correspond to the same path in the control flow graph.

In order to be able to compute the appropriate alias relation in such cases, the memory map must be enriched in the following way. Firstly, all assignments are numbered in order to deduce their order and the current symbolic value of each variable is annotated with the number of the assignment which set this value. Secondly, in the case of an assignment in which the symbolic access path of the variable is not constant, the variable to update in the memory map is found by evaluating the symbolic access path for this test case by substituting in the current values of the input variables. Once a variable has been assigned a value using an access path which had to be evaluated in this way, the memory map stores for this variable not only the current symbolic value, but also all the symbolic access paths used in this and previous assignments, each annotated with the number of the most recent assignment using this symbolic access path.

Throughout substitution, the current symbolic values of variables are read from the memory map using symbolic access paths which may also not be constant. In these cases too, the symbolic access path is evaluated by substituting in the current values of the input variables. When the current symbolic value of any variable is read from the memory map, it is conditioned by the following alias relation on the symbolic inputs : the symbolic access path of the variable used to read its value is equal to the symbolic access path used in the assignment of the current value and is not equal to the symbolic access paths used in all subsequent assignments (because no subsequent assignment updated the value of this variable). Note that if an input variable whose value has not been updated is referenced using a symbolic access path which depends on the value of other input variables, then there was no assignment of the current value and the alias relation is just the difference between the symbolic access path used to reference the variable and those used in all assignments until this point. The different alias relations generated when a symbolic value is read from the memory map are ordered according to their annotation, with the alias relation originating in the most recent assignment coming first.

In the example of Figure 3, the treatment of the first element of the symbolic execution path, corresponding to the first assignment in this function results in the symbolic value *x**2, annotated with the number 1, being set for intermediate variable `v` in the memory map. The second instruction to be executed causes the symbolic value of `v` to be updated to *x**2 - *y*, annotated with the number 2. After treatment of the 3rd assignment, the symbolic value of `pt` is set to `tab` + 2, annotated with the number 3 and the 4th assignment causes the symbolic value of `*(tab + 2)` to be set to *x*, annotated with the number 4. Up until this point, substitution is common to all test cases. However, the 5th assignment is to a variable referenced by an access path depending on input values. Suppose that in the test case whose symbolic execution path is being treated, *x* = -2 and *y* = -3. Then *(tab + 2 + *x**2 - *y*) evaluates to `*(tab + 1)` and the symbolic value of *y* is added to the memory map for `*(tab + 1)`, annotated with the number 5 and a symbolic access path of *(*tab* + 2 + *x**2 - *y*) is also added for `*(tab + 1)`, also annotated with the number 5. To treat the conditional instruction, *(tab + *y* + 4) is evaluated to `*(tab + 1)` and the symbolic value of `*(tab + 1)` is read from the memory map as being *y* if *(*tab* + *y* + 4) = *(*tab* + 2 + *x**2 - *y*). The term of the disjunctive path predicate covered by this test case is thus:

	$y + 4 = 2 + x*2 - y$	(alias relation)
$\wedge$	$y =< 5$	(failure 1st condition).

However, test input values -2 for *x* and *y* cause the memory map to be updated differently for the 5th assignment. This time it is `*(tab + 0)` which receives a symbolic value of *y* and a symbolic access path of *(*tab* + 2 + *x**2 - *y*), annotated with the number 5. To treat the conditional instruction, *(tab + y + 4) is evaluated to `*(tab + 2)` and the symbolic value of `*(tab + 2)` is read from the memory map as being *x* if *(*tab* + *y* + 4) = *(*tab* + 2) and *(*tab* + *y* + 4) ≠ *(*tab* + 2 + *x**2 - *y*). These conditions are inverted because of their respective numbers and the "path predicate" is now:

	$y + 4 \neq 2 + x*2 - y$	(alias relation)
$\wedge$	$y + 4 = 2$	(alias relation)
$\wedge$	$x =< 5$	(failure 1st condition).

Finally, if *x* is –2, *y* is –4 and *tab*[0] is -9 on input, then the "path predicate" becomes:

	$y + 4 \neq 2 + x*2 - y$	(alias relation)
$\wedge$	$y + 4 \neq 2$	(alias relation)
$\wedge$	$tab[(y + 4)] =< 5$	(failure 1st condition).

This treatment of aliases does not introduce an overhead (except for the numbering of assignments) if it is not needed. If a program only uses access paths which do not depend on input values then the alias relation reduces to true and no information needs to be stored in the memory map other than the current symbolic values of the variables, annotated with the assignment number. In this case, the number of symbolic expressions stored in the memory map is bounded by the number of variables to which values are assigned in the execution path (itself bounded by the number of assignments in the path). Each symbolic expression stored can, of course, be arbitrarily complicated.

If the program does use access paths depending on input values then the extra information to be stored in the memory map for each variable involved is bounded by the product of the number of assignments and the number of dimensions of the data structures concerned. This is because a C access path can be indexed by a number of variable expressions which is bounded by the number of dimensions of the data structure.

It is easier to present substitution and test selection as successive phases but in our prototype implementation they are in fact interleaved. This enables us to use the constraint solver's backtracking mechanism to recover the memory map of any path prefix which is shared by some previous path. This avoids numerous recalculations of the same memory map state. However, the state of the memory map cannot be shared in this way if it has been enriched in order to calculate alias relations, i.e. for prefixes containing access paths depending on input values.

6. Test Selection and Constraint Solving

We explained in Section 3 how our "on-the-fly" test generation strategy is initiated by the generation of a first test case by picking input values inside the input domain *ID* of the program under test. *ID* is defined by a Cartesian product of the domains of the input parameters and a conjunction *CID* of constraints reflecting input parameter dependencies and the input values of the first test case are found by solution of this constraint satisfaction problem. For the subsequent test cases, we do not (as in most other approaches) select the next path to be tested, but the next path prefix. This is characterised by a path predicate prefix, which is also solved by constraint solving to obtain the next test case. Let us start by describing how we select the successive path predicate prefixes, before a few remarks about our strategy for constraint solving.

The first test case t_1 is generated from a selection domain SD_0 which is the input domain *ID*. From the execution of t_1, we derive the corresponding path

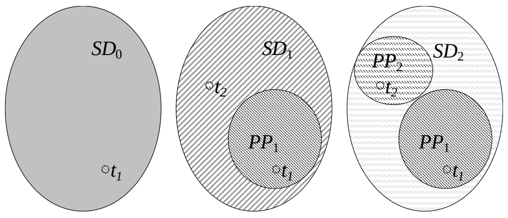

Figure 4 : input domains

predicate PP_1. In order to cover a new path, we have to generate test inputs from the difference of SD_0 and the domain of PP_1, let us note SD_1 this selection domain (see Figure 4). If SD_1 is empty, this means that there are no more paths to cover.

Otherwise, we can generate a new test case t_2, from SD_1, which exercises a new path whose predicate is PP_2. This process is repeated until an empty selection domain SD_n is reached, in which case we have covered every feasible path of the program under test. To sum up this selection strategy, at each step *i*, we generate a test t_i from a selection domain SD_i, and when executed, this test exercises a path whose predicate is PP_i. Let us assume that there exist *n* feasible paths, and let us call CSD_i the constraints defining SD_i, then each CSD_i can be recursively defined in the following way:

$CSD_0 = CID$

$\forall\ i \in 1..n,$

$CSD_i = CSD_{i-1} \wedge \neg PP_i = CID \wedge \neg PP_1 \ldots \wedge \neg PP_i$

Note that each path predicate PP_i is the ordered conjunction of the number p_i of successive conditions C_i^j encountered along the corresponding path:

$PP_i = C_i^1 \wedge \ldots \wedge C_i^{pi}$

An ordered case analysis of the negation of each condition C_i^j shows that the negation of PP_i is just the disjunction of all the prefixes of PP_i with the last condition negated :

$$\neg PP_i = \neg C_i^1 \vee \bigvee_{m=2..\,pi} (C_i^1 \wedge \ldots \wedge \neg C_i^{m-1} \wedge \neg C_i^m)$$

Note that each term of this disjunction is a conjunction of conditions corresponding to a (possibly infeasible) path prefix in the control flow graph which is unexplored at the *i*th step of our selection strategy. Let us consider the longest feasible conjunction $MaxC_i$. We choose to generate the next test case t_{i+1} from the domain of $MaxC_i$. This has two important effects : the path predicate PP_{i+1} of t_{i+1} contains $MaxC_i$ as prefix and the negation of PP_{i+1} (expressed in the above form) subsumes the negation of all previous paths.

Indeed, the longest feasible conjunction $MaxC_{i+1}$ in $\neg PP_{i+1}$ contains all the conditions with unexplored alternatives from path predicates PP_1 to PP_i. These alternatives can be seen as choice points in the search for a solution. Our strategy corresponds in this sense to a depth-first construction of the graph of the feasible paths in the control flow graph. Choice points are placed on each condition encountered and when all the choice points have been explored, there are no more feasible paths to cover.

Note that an advantage of our path predicate prefix selection strategy is that the depth of the constraint solver stack is bounded by the number of conditions (and alias relations) in the longest feasible path.

To respect the *k*-paths criterion, the definition of $MaxC_i$ must be modified to take into account the annotations of conditions from the heads of loops with a variable number of iterations. If a condition corresponds to exit from the loop after *k* iterations and its negation would ensure definitive loop re-entry (see Section 4), or if it corresponds to exit or re-entry after more than *k* iterations, then the alternative of this condition is not explored, because they will certainly give rise to paths containing more than *k* iterations of this loop. We cannot prevent constraint solving of some path predicate prefix resulting in a path which, after the prefix, executes more than *k* iterations of a loop. However, our strategy does ensure that we never generate any new path predicate prefixes containing too many loop iterations.

Now let us consider the nature of the constraints to be solved and our strategy for constraint solving. The first constraint solution problem is defined by the input domain of the C program under test. Note that the formal parameters of a C function may not all be input parameters and that some global variables may also be input parameters. The parameters may be accessed via pointers or belong to structured data types of possibly unknown dimensions. In our example, `t3` is not an input parameter and the sizes of `t1` and `t2` are variable. We therefore currently ask the user to define the input parameters and their domains. In the case of structured data, the user must define the domain of any variable dimensions and of each component (elements, fields, de-referenced values,…) of the input parameters. Note that, by default, the domains of scalar input parameters and of input parameter components can be set to the whole of their type in C. More restricted domains may be expressed as an interval or finite set of values. A limited form of universal quantification can be used to define the domains of indexed components. We also ask the user to define the input parameter dependencies, *CID*, also using a limited form of universal quantification if necessary. In our example (see Figure 5), the input parameter `l1` is used in the loops to limit the values of the index when accessing elements of `t1` (and similarly for `l2` and `t2`). The value of `l1` must therefore be less than or equal to the length of `t1` (and in fact, we set them as equal to simplify our presentation). Furthermore, the elements of `t1` must be ordered.

```
dim(t1) ∈ 0..10000
l1 ∈ 0..10000
forall i ∈ 0..dim(t1). t1[i] ∈ -100..100
dim(t1) = l1
forall i ∈ 1..l1. t1[i] ≥ t1[i - 1]

and similarly for t2
```

Figure 5 : domains and *CID* for Merge

Universally quantified domains and constraints are handled as follows during constraint solving: in the case of data-structures whose size may not be the same in all the test cases, constrained variables representing the elements of the data-structure are defined only "as needed". That is to say: if at some point during constraint solving the lower limit on the domain of the size of the data-structure exceeds zero, or is increased, constrained variables are created to represent the minimum number of elements of the data-structure. Moreover, when (un-quantified) constraints on specific elements are applied, then the size of the data-structure is automatically constrained to be at least large enough to include these elements. When a constrained variable representing an element of such a data-structure is created, the domain definition, as well as any constraints in *CID* which are universally quantified over an index of elements of this data-structure, are instantiated and applied to this element. In our example, adding the first path condition encountered, 0 < `l1`, to the constraint *dim*(`t1`) = `l1` from *CID* implies 0 < *dim*(`t1`). This leads to creation of a new constrained variable to represent `t1[0]` with an initial domain defined by instantiation of the quantified domain definition

forall i ∈ 0..*dim*(`t1`). `t1`[i] ∈ -100..100

but the quantified constraint from *CID*

forall i ∈ 1..`l1`. `t1`[i] ≥ `t1`[i - 1]

cannot be instantiated for t1[0].

The constraints to be solved at any point are a conjunction of Boolean and arithmetic constraints. The satisfiability of such constraints over finite intervals is decidable and we use constraint logic programming techniques to compute their solutions. They are implemented in the Eclipse CLP environment [20] and adapted from the constraint solving procedure used in [8] and [14]. For constraints on floating-point numbers, we currently use the incomplete procedure offered by the Eclipse Interval Constraint library.

We use the following "labelling heuristics" [4]: we search for values for constrained variables other than those representing dimensions of data-structures using a random generator, biased towards the middle of the current domain of the variable, after constraint propagation.

However, our search strategy for dimension values is to choose them as low as possible. This has the advantage that we are sure to generate tests for empty data-structures, where they are allowed, whose treatment is often a source of bugs. Moreover, this strategy reduces execution time because fewer constrained variables representing data-structure elements are generated and processed. A further advantage is that there is often a link between data-structure dimensions and the number of loop iterations which means that smaller data-structures result in fewer superfluous test cases for the *k*-path criterion.

An interesting feature of our test generation strategy is that we only analyse feasible path predicates. Of course during the search for $MaxC_i$, we may construct other path predicate prefixes which turn out to be unsatisfiable, but this is always due to the negation of the last condition. This kind of unsatisfiability is easier to detect than that due to the structural construction of arbitrary path predicates. Moreover, when a path predicate prefix has no solution, the strategy does not construct or explore any path predicates starting with this prefix.

7. Example

Now let us follow step by step what happens when we run the prototype implementation of our test generator on the example function Merge. Let us start by examining the control flow graph of Merge, shown in Figure 6 (in which the assignments to `t3` are left out for simplicity). Many theoretical paths through the graph are in fact infeasible: the first loop "consumes" the whole of `t1` or `t2` and zero or more elements of the other array, but not all of them. The second loop consumes those elements of `t1` not consumed by the first loop and the third loop does the same for `t2`. This means that the number of iterations of either the second or third loop is zero in all feasible paths and that if the first loop has at least one iteration then so will either the second or third. Moreover, it can easily be seen from Figure 6 that paths containing fragments such as …,2,7,… are infeasible.

Finally, remember that we have set *k*, the maximum number of iterations of any loop, to 2. The number of theoretical paths therefore becomes 126, of which just 17 are feasible.

The first test case for our example, shown in Table 1 is generated from the domains and constraints of Figure 5. Note that we have used relatively limited domains for the elements of `t1` and `t2` only to simplify the presentation; enlarging the domains does not alter the execution time in this example. Our search strategy for dimension values (described in Section 6) results in the sizes of `t1` and `t2` being set to zero in the first test case of our example. The predicate PP_1 of the path covered by the first test encounters the following conditions, shown in

Figure 7:

- $C_1^1 = Cond_1$: $\neg\ 0 <$ `l1` (exit loop number 1 after 0 iterations: arc 2 in Figure 6)
- $C_1^2 = Cond_2$: $\neg\ 0 <$ `l1` (exit loop number 2 after 0 iterations: arc 8)
- $C_1^3 = Cond_{3:}$ $\neg\ 0 <$ `l2` (exit loop number 3 after 0 iterations: arc 10)

From PP_1, we derive $MaxC_2 = Cond_1 \wedge Cond_2 \wedge \neg Cond_3$.

Solution of $MaxC_2$ generates the second test case shown in Table 1, in which t2 has one element and there is one iteration of the third loop only. The third test, in which t1 is still empty and t2 has two elements, resulting in two iterations of the third loop, is generated in a similar way.

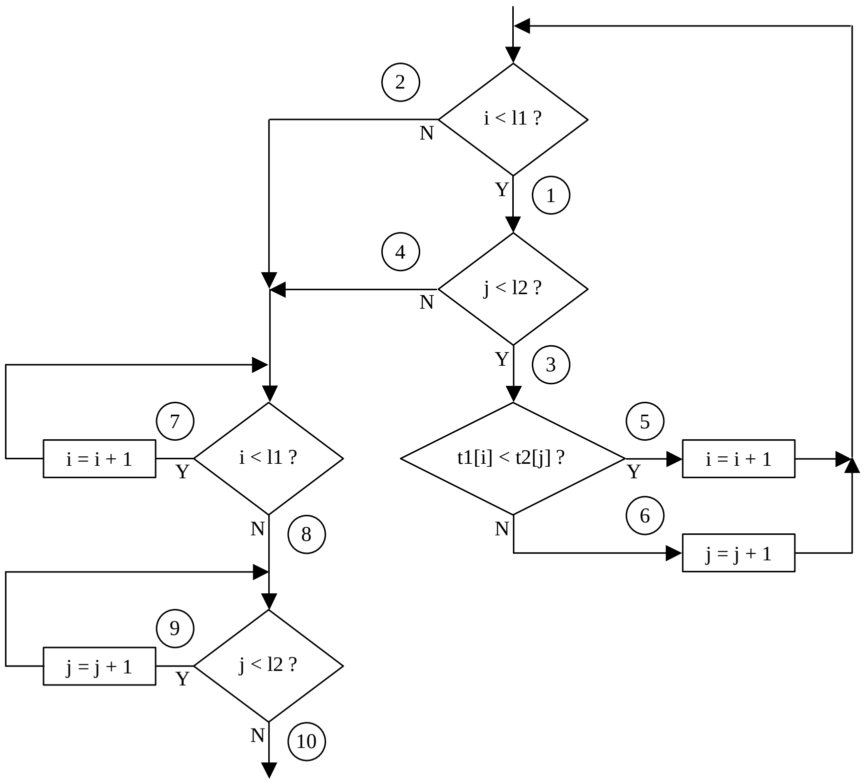

Figure 6 : control flow graph of Merge with multiple condition decomposed

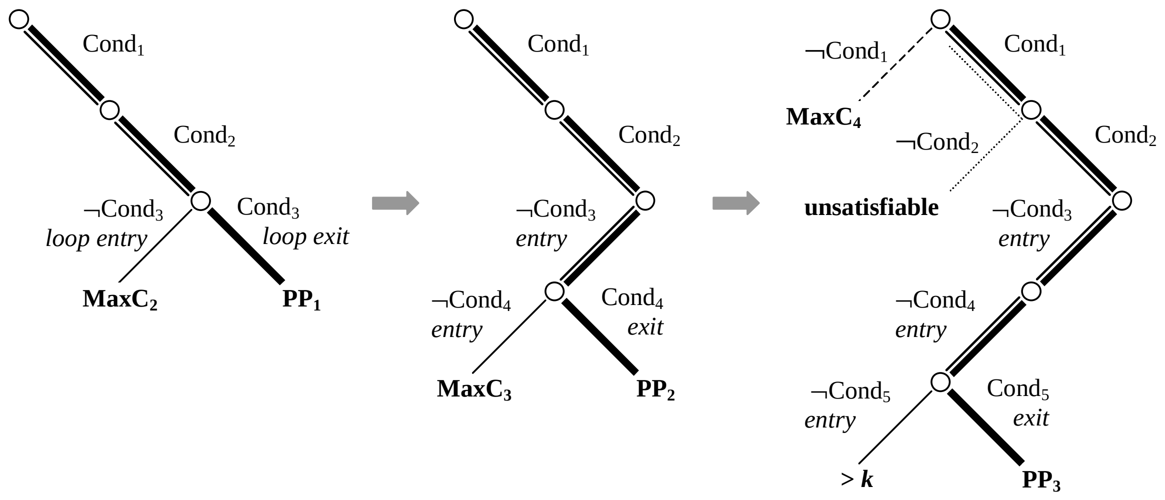

Figure 7 : test selection strategy for Merge

With no limit on loop iterations, $MaxC_4$ would be the conjunction of :

- $C_3^1 = Cond_1$: $\neg\ 0 <$ `l1` (exit 1st loop after 0 iterations: arc 2)
- $C_3^2 = Cond_2$: $\neg\ 0 <$ `l1` (exit 2nd loop after 0 iterations: arc 8)
- $C_3^3 = \neg Cond_3$: $0 <$ `l2` (entry 1st iteration of 3rd loop: arc 9)
- $C_3^4 = \neg Cond_4$: $1 <$ `l2` (entry 2nd iteration of 3rd loop: arc 9)
- $\neg C_3^5 = \neg Cond_5$: $2 <$ `l2` (entry 3rd iteration of 3rd loop: arc 9)

This is where the modification of our strategy to limit loop iterations takes effect: although this conjunction is satisfiable, it is not solved because it would entail more than 2 iterations of the third loop. Our strategy thus backtracks to the lowest unexplored branch and constructs the path prefix $Cond_1 \wedge \neg Cond_2$. However, this is unsatisfiable, so $MaxC_4$ is in fact $\neg Cond_1$. Test generation continues until the solution of $MaxC_{12}$, the predicate of the path prefix 1,3,5,1,3,5,1, results in the 12th test covering the path 1,3,5,1,3,5,1,3,6,1,4,7,8,10, containing three iterations of the first loop. At this point, exploration of the predicates of all other paths starting with 1,3,5,1,3,5,1,3 is blocked by the limit on the number of loop iterations, and the path prefix 1,3,5,1,3,5,1,4 is constructed. This contains the infeasible fragment 3,5,1,4, as does the path 1,3,5,1,4 to the next deepest unexplored branch, so $MaxC_{13}$ is in fact the predicate of the path prefix 1,3,6.

We cannot entirely avoid generation of superfluous tests (of the 19 tests generated in our example, the 12th and 17th contain more than 2 iterations of the same loop) but our strategy does enable us to prune the search space effectively once a superfluous test is generated. Pruning of infeasible paths is effective too: in this example, we only need to discover the infeasibility of 25 path predicate prefixes in order to eliminate the 109 infeasible paths.

To test the stability of our results, we ran our prototype generator ten times on the Merge example with k set to 5 and maximal domains for the elements of `t1` and `t2`. The number of theoretical paths is 4536, of which 321 are feasible. 337 tests were generated in each run and 317 infeasible path predicate prefixes found in order to eliminate the 4215 infeasible paths. The CPU execution times in seconds on a 2GHz PC running under Linux were:
2.06, 2.08, 2.04, 2.05, 1.98, 2.09, 1.96, 2.04, 2.06, 2.02.

These results are encouraging : our labelling heuristics seem to be effective as 654 predicate prefixes were generated and solved or rejected in just 2 seconds. This includes execution of the instrumented Merge function and many file operations for communication and trace purposes. A few obvious optimisations of our implementation would certainly allow us to obtain even shorter execution times. Our random labelling heuristic did not feed through to any substantial variation in execution times, which suggests that it does not take up a disproportionate amount of the execution time.

8. Further work

We have already illustrated, in Section 6, some ways in which our basic test generation method can be modulated in order to implement variations in the test selection strategy. Firstly, constraints other than those from a path predicate can be taken into account, as is already done for the treatment of the precondition on the input values of the program under test. Other external constraints could be derived from a more complete specification of the

Table 1 : tests generated for Merge

test no.	l1	l2	t1[0]	t1[1]	t1[2]	t2[0]	t2[1]	t2[2]	path covered (with selected prefix underlined)
1	0	0							2,8,10
2	0	1				-3			2,8,9,10
3	0	2				-52	30		2,8,9,9,10
4	1	0	-5						1,4,7,8,10
5	2	0	-41	-8					1,4,7,7,8,10
6	1	1	-17			16			1,3,5,2,8,9,10
7	1	2	24			67	88		1,3,5,2,8,9,9,10
8	2	1	-67	14		-22			1,3,5,1,3,6,1,4,7,8,10
9	3	1	-77	-27	0	-61			1,3,5,1,3,6,1,4,7,7,8,10
10	2	1	-1	23		46			1,3,5,1,3,5,2,8,9,10
11	2	2	-68	-37		-14	29		1,3,5,1,3,5,2,8,9,9,10
12	3	1	-69	-36	28	-5			1,3,5,1,3,5,1,3,6,1,4,7,8,10
13	1	1	-23			-50			1,3,6,1,4,7,8,10
14	2	1	41	73		9			1,3,6,1,4,7,7,8,10
15	1	2	-30			-69	24		1,3,6,1,3,5,2,8,9,10
16	1	3	-30			-73	-13	15	1,3,6,1,3,5,2,8,9,9,10
17	2	2	31	56		-17	64		1,3,6,1,3,5,1,3,5,2,8,9,10
18	1	2	27			-54	-26		1,3,6,1,3,6,1,4,7,8,10
19	2	2	-52	-26		-79	-65		1,3,6,1,3,6,1,4,7,7,8,10

program under test or from integration testing scenarios or test objectives. Complementary, approximate, static analyses could be used to reduce the input domain in the case of test objectives which are not expressed in terms of input values. Secondly, information collected during execution of the program under test can also influence test selection, as illustrated by the use of annotations of loop-head conditions to implement the *k*-path criterion. We currently treat calls to other non-recursive functions by classic in-lining techniques. By annotating the conditions in called functions, the exploration of different paths in these functions could be restricted.

The effectiveness of our test generation strategy is limited by the selection of a single test for each path. Our strategy is weaker on the detection of bad positioning of path domain boundaries than domain testing [10]. Our approach could be modified to select tests at the limits of the path domain boundaries, where bugs are often found, at the cost of a linear increase in the number of tests. The generation of a single test per path also limits the chances of detecting coincidental correctness, unlike statistical structural testing [8], but we could randomly generate several tests for each path.

Apart from certain optimisations of the current prototype, further work will therefore be devoted to the investigation of modifications of our structural test generation method so as to include it in an overall strategy which would reduce the number of redundant tests and increase the chances of detecting bugs. We already take into account a pre-condition on input values but the oracle must currently be hand-coded. By taking certain forms of post-condition on the C variables into account, we could automatically generate the oracle. Moreover, we could use such constraints derived from the specification to replace function calls and implement a structural integration testing strategy. By avoiding full structural testing of the call graph, this would enable us to apply our approach to integration testing of realistic programs.

References

[1] P. Baudin, A. Pacalet, J. Raguideau, D. Schoen and N. Williams, *CAVEAT: A Tool for Software Validation*, International Performance and Dependability Symposium, Washington D.C., June 2002

[2] B. Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, New York, NY, 1990

[3] G. Bernot, M.C. Gaudel and B. Marre, *Software Testing based on Formal Specifications: a theory and a tool*, Software Engineering Journal, Vol. 6, No. 6, pp 387-405, Nov. 1991

[4] A M Cheadle, W Harvey, A J Sadler, J Schimpf, K Shen and M G Wallace, *ECLiPSe: An Introduction (chapter 12)*, IC-Parc, Imperial College London, Technical Report IC-Parc-03-1, 2003

[5] M.J. Gallagher and V.L. Narasimhan, *ADTEST : A Test Data Generation Suite for Ada Software Systems*, IEEE

Transactions on Software Engineering, Vol. 23, No. 8, August 1997

[6] A. Gottlieb, B. Botella and M. Reuher, *A CLP Framework for Computing Structural Test Data*, CL2000, LNAI 1891, Springer Verlag, July 2000, pp 399-413

[7] E. Goubault, A. Pacalet, B. Starynkévitch, F. Védrine and D. Guilbaud, *A Simple Abstract Interpreter for Threat Detection and Test Case Generation*, WAPATV'01, Toronto, Canada, May 2001

[8] S-D Gouraud, A. Denise, M-C. Gaudel and B. Marre, *A New Way of Automating Statistical Testing Methods*, ASE 2001, Coronado Island, California, November 2001

[9] N. Gupta, A.P. Mathur and M.L. Soffa, *Generating Test Data for Branch Coverage*, ASE 2000, Grenoble, France, Sep. 2000

[10] B. Jeng and E.J. Weyuker, *A Simplified Domain-Testing Strategy*, ACM Transactions on Software Engineering and Methodology, Vol. 3, No. 3, July 1994, pp 254-270

[11] B. Korel, *Automated Software Test Data Generation*, IEEE Transactions on Software Engineering, Vol. 16, No. 8, August 1990

[12] G.T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D.R. Cok, *How the Design of JML Accommodates Both Runtime Assertion Checking and Formal Verification*, In F.S. de Boer, M.M. Bonsangue, S. Graf, and W-P. de Roever (eds.), Formal Methods for Components and Objects, pages 262-284. LNCS vol. 2852, Springer Verlag, Berlin, 2003

[13] Y. Ledru, L. du Bousquet, O. Maury and P. Bontron, *Filtering TOBIAS Combinatorial Test Suites*, ETAPS/FASE 2004, Barcelona, Spain, March/April 2004

[14] B. Marre and A. Arnould, *Test sequences generation from Lustre descriptions: GATeL*, ASE 2000, Grenoble, pp 229--237, Sep. 2000

[15] C. Michael and G. McGraw, *Automated Software Test Data Generation for Complex Programs*, ASE, Oct 1998, Honolulu

[16] C. Michel, M. Rueher and Y. Lebbah, *Solving Constraints over Floating-Point Numbers*, CP'2001, LNCS vol. 2239, pp 524-538, Springer Verlag, Berlin, 2001

[17] A.J. Offut, Z. Jin and J. Pan, *The Dynamic Domain Reduction Procedure for Test Data Generation*, Software Practice and Experience, Vol. 29, No. 2, pp 167-193, Jan 1999

[18] R.E. Prather and J.P. Myers, *The Path Prefix Testing Strategy*, IEEE Transactions on Software Engineering, Vol. 13, No. 7, July 1987

[19] N.T. Sy and Y. Deville, *Consistency Techniques for Interprocedural Test Data Generation*, ESEC/FSE'03, September 1-5, 2003, Helsinki, Finland

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