PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis

Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger

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

Abstract. We present the PathCrawler prototype tool for the automatic generation of test-cases satisfying the rigorous all-paths criterion, with a user-defined limit on the number of loop iterations in the covered paths. The prototype treats C code and we illustrate the test-case generation process on a representative example of a C function containing data-structures of variable dimensions, loops with variable numbers of iterations and many infeasible paths. PathCrawler is based on a novel combination of code instrumentation and constraint solving which makes it both efficient and open to extension. It suffers neither from the approximations and complexity of static analysis, nor from the number of executions demanded by the use of heuristic algorithms in function minimisation and the possibility that they fail to find a solution. We believe that it demonstrates the feasibility of rigorous and systematic testing of sequential programs coded in imperative languages.

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.

2 Related Work

There has been much research on the automatic generation of structural test-cases but most of it addresses the Test Data Generation Problem (TDGP) of finding data to cover a test objective in the form of given node, branch or path of the control flow graph.

M. Dal Cin, M. Kaâniche, and A. Pataricza (Eds.): EDCC 2005, LNCS 3463, pp. 281–292, 2005.
© Springer-Verlag Berlin Heidelberg 2005

```
void Merge (int t1[],int t2[],int t3[],int l1,int l2){  (1
  int i = 0;   int j = 0;   int k = 0;                   (2
  while (i < l1 && j < l2) {                             (3
    if (t1[i] < t2[j]) {                                 (4
      t3[k] = t1[i];                                     (5
      i++; }                                             (6
    else {                                               (7
      t3[k] = t2[j];                                     (8
      j++; }                                             (9
    k++; }                                               (10
  while (i < l1) {                                       (11
   t3[k] = t1[i];                                        (12
   i++;                                                  (13
   k++; }                                                (14
  while (j < l2) {                                       (15
   t3[k] = t2[j];                                        (16
   j++;                                                  (17
   k++; }                                                (18
}                                                        (19
```

Fig. 1. Source code of the function `Merge`

Static approaches to test-case generation [2, 3, 15] typically select a path from the control flow graph covering the test objective, derive the path predicate as a set of constraints on the input values 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.

Dynamic approaches [1, 5, 9] avoid the problems of symbolic execution by dispensing with the path predicate and using general heuristic function minimisation techniques to modify the input data so that the test objective is covered. The first set of input data is arbitrarily selected and the program is instrumented so as to indicate the branches taken and evaluate their "distance" from the test objective. Function minimisation must reduce this distance to zero. 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.

We address a different problem to that of most previous work, and adopt a different solution. We believe that rigorous testing is possible if sufficiently automated and we therefore base our work on a rigorous test criterion: 100% coverage of feasible execution paths. The TDGP is not the best formulation of this problem. We do not need to construct the control flow graph, enumerate all the paths in the graph, many of which will be infeasible, and search for a test for each. Instead, we iteratively cover "on the fly" the whole input space of the program under test. This is an extension of the idea sketched out in [14].

Like the dynamic approaches to test data generation, PathCrawler is based on dynamic analysis, but instead of heuristic function minimisation, it uses constraint logic programming to solve a (partial) path predicate and find the next test-case, as in the approaches based on static analysis. It suffers neither from the approximations and complexity of static analysis, nor from the number of executions demanded by heuristic algorithms used in function minimisation and the possibility that they fail to find a solution.

3 Our Approach

Our approach is applicable to all sequential programs coded in an imperative language and the prototype has been implemented for C. This paper extends [8], notably by illustrating the test generation process step-by-step on an example: the C function `Merge`, whose source code is shown in Fig. 1. `Merge` takes as input two arrays, `t1` and `t2`, of ordered integers and their effective lengths, `l1` and `l2`, and outputs, in array `t3`, all their elements, in order. `Merge` is representative of many of the problems posed by C code: it contains arrays of variable length, loops with a variable number of iterations and many infeasible paths and only produces the correct result if the input arrays are sorted. In this section we give an overview of our approach and in the following sections we describe its principal stages: Instrumentation, Substitution and Constraint Solving. We then describe the results of applying PathCrawler to the function `Merge`, before concluding with a discussion of further work.

Our approach (see Fig. 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 which 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

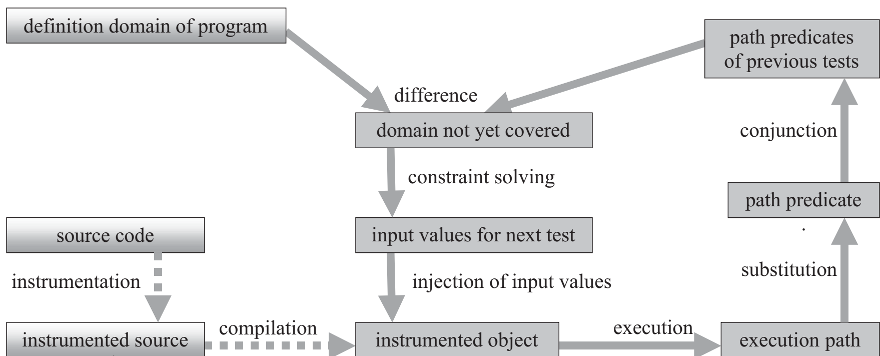

Fig. 2. Our approach

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.

Loops with a variable number of iterations, such as the three loops in our example function `Merge`, can cause a combinatorial explosion in the number of execution paths. 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. 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 symbolic execution path, i.e. a sequence of assignments and satisfied conditions on C variables or access paths. A trace instruction is therefore inserted after each assignment and each branch of the source code. The instrumentation is implemented using the CIL library [12]. Certain source-code statements are decomposed, notably multiple conditions which reinforces our test criterion, bringing it close to all-paths combined with MC/DC. Note that in the C language, variable values may be referenced using "data access paths" involving not only the operators to access array elements or structure fields, but also pointer de-references. In our trace instructions, all data access paths are rewritten, in a purely syntactic transformation, to a canonical form, so as to simplify the substitution stage.

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 [2] and can treat aliases (two or more ways of denoting the same memory location) with relative ease.

6 Test Selection and Constraint Solving

The first test-case t_1 is generated from a selection domain SD_0 which is the input domain, ID, of the program under test. ¿From the execution of t_1, we derive the corresponding path predicate PP_1. In order to cover a new path, we have to generate test inputs from the difference, SD_1, of SD_0 and the domain of PP_1 (see Fig. 3). 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.

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:

$$PP_i = C_{i,1} \wedge \ldots \wedge C_{i,pi} \tag{1}$$

The negation of PP_i is just the disjunction of all the prefixes of PP_i with the last condition negated :

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

Note that each term of this disjunction is a conjunction of conditions corresponding to a (possibly infeasible) path prefix which is unexplored at the ith step of our selection strategy. To find a solution in each selection domain SD_i, we choose to solve the longest feasible conjunction in $\neg\, PP_i$, which we call $MaxC_i$. If all the conjunctions in $\neg\, PP_i$ are infeasible, the longest unsolved feasible conjunction in $\neg\, PP_{i-1}$, $MaxC_{i-1}$, is tried, and so on. Our strategy corresponds in this sense to a depth-first construction of the tree of feasible execution paths, as we will illustrate below on our example function.

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 the negation of a condition would result in loop re-entry after k or more iterations, then it is not explored. This way, we ensure

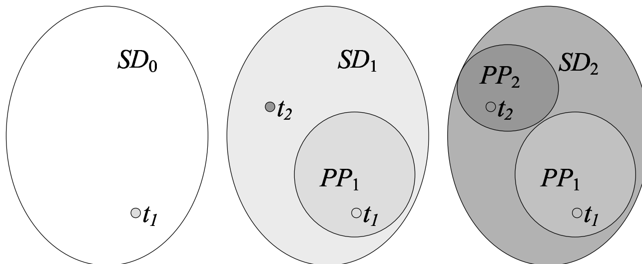

Fig. 3. Input domains

that we never generate any new path predicate prefixes containing too many loop iterations. However, we cannot prevent constraint solving of some path predicate prefix occasionally resulting in a "superfluous" test, i.e. one covering a path which - after the prefix - executes more than k iterations of a loop.

Test selection and constraint solving are implemented in the Eclipse constraint logic programming environment [17]. Note that solving non-linear constraints is decidable only for data types with finite domains, such as integers. However, current research [10, 15] holds the promise of decidable and precise constraint solving for floating-point numbers too. Solving constraints over finite domains is NP-complete in the worst case but we base our work on heuristics developed for test-case generation problems [3, 7] which display low complexity in practice. 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. Our "labelling" heuristic (used to generate and test values after constraint propagation) is to choose dimension values 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, as there is often a link between data-structure dimensions and the number of loop iterations, smaller data-structures can result in fewer superfluous test cases for the k-path criterion. For variables other than dimensions, labelling uses a random generator which starts by generating values in the median third of the variable's domain after constraint propagation. If all these values have been tried without success, randomly generated values outside the median third are tried.

An advantage 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 PathCrawler on our example. The first step is to define the input domain, ID, of `Merge`. 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. PathCrawler can treat functions with pointers and structured data, including arrays with variable dimensions, as input parameters. However, as the input parameters of a C function cannot be automatically identified without static analysis, the user is currently asked to pick out the input parameters from the list of all the scalar formal parameters and global variables visible to

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

Fig. 4. Merge domains and preconditions

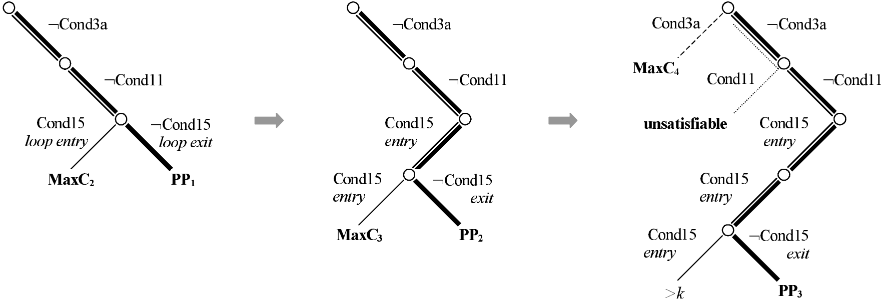

Fig. 5. Selection strategy

the function, and of all the components (elements, fields, de-referenced values,) of structured formal parameters and visible global variables, or those in the form of pointers. The user is also asked to replace the default domain (based on its declared C type) of each input parameter by a smaller interval, when applicable. Similarly, the user must give the upper limit of any variable dimensions of arrays containing input parameters. Finally the user must define any other input parameter dependencies (precondition). The $forall$ operator, which iterates over all elements of an array, can currently be used in the precondition definitions and we are studying the use of a richer language to specify the precondition. In our example (see Fig. 4), the value of `l1` (resp. `l2`) must be less than or equal to the length of `t1` (resp. `t2`) (and in fact, we set them as equal). Furthermore, `t1` and `t2` must be ordered.

In the first test-case of our example, generated from the domains and constraints of Fig. 4, the sizes of `t1` and `t2` are set to zero. This test-case is shown in Table 1, in which the arcs of the execution path are denoted by the line-number of the corresponding condition in the source code (in Fig. 1), preceded by a minus sign if the condition is not satisfied and, in the case of composite conditions, followed by a letter indicating the sub-condition concerned. The predicate PP_1 of the path covered by the first test encounters the following conditions (also numbered according to their origin in the source code), shown in Fig. 5:

$C_{1,1} = \neg\ Cond3a : \neg\ 0 < l1$ (exit 1st loop after 0 iterations)
$C_{1,2} = \neg\ Cond11 : \neg\ 0 < l1$ (exit 2nd loop after 0 iterations)
$C_{1,3} = \neg\ Cond15 : \neg\ 0 < l2$ (exit 3rd loop after 0 iterations)

Table 1. Tests generated for `Merge`

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

Solution of $MaxC_2 = \neg\, Cond3a \wedge \neg\, Cond11 \wedge Cond15$ generates the second test-case shown in Table 1, in which there is one iteration of the third loop. The third test, covering two iterations of the third loop, is generated in a similar way. With no limit on loop iterations, $MaxC_4$ would be:

$$\begin{aligned} C_{3,1} &= \neg\, Cond3a : \neg\ 0 < l1 \text{ (exit 1st loop after 0 iterations)} \\ C_{3,2} &= \neg\, Cond11 : \neg\ 0 < l1 \text{ (exit 2nd loop after 0 iterations)} \\ C_{3,3} &= \quad Cond15 : \quad 0 < l2 \text{ (entry 1st iteration of 3rd loop)} \\ C_{3,4} &= \quad Cond15 : \quad 1 < l2 \text{ (entry 2nd iteration of 3rd loop)} \\ \neg\, C_{3,5} &= \quad Cond15 : \quad 2 < l2 \text{ (entry 3rd iteration of 3rd loop)} \end{aligned}$$

This is where the modification of our strategy to limit loop iterations takes effect: this conjunction is not solved because it would entail more than 2 iterations of the third loop. Our strategy thus backtracks to the lowest unexplored branch of Fig. 5 and constructs the path prefix $\neg\, Cond3a \wedge Cond11$. However, this is unsatisfiable, so $MaxC_4$ is in fact $Cond3a$.

Of the 19 tests generated in our example, only the 12th and 17th are superfluous (contain more than 2 iterations of the same loop) and we only need to discover the infeasibility of 25 path predicate prefixes. In comparison, `Merge`'s control-flow graph contains 109 infeasible paths if k is set to 2.

To test the efficiency and stability of our implementation, we ran our prototype ten times on `Merge` with k set to 5 and maximal domains for the elements of `t1` and `t2`. For this value of k, the control-flow graph contains 4536 paths, of which 321 are feasible. In each run, 337 tests were generated and 317 infeasible path predicate prefixes found in order to eliminate the 4215 infeasible paths, i.e. 654 predicate prefixes were generated and solved or rejected. The CPU execution time on a 2GHz PC running under Linux varied between 0.75 and 0.81

seconds, with an average of 0.785. In conclusion, all k-paths were tested (and 20 superfluous tests generated) in under 1 second and our random labelling heuristic did not cause much variation in execution time. For $k = 10$, 20993 tests are generated and 15357 infeasible paths eliminated in around 116 seconds.

8 Other Examples

We have also tried our prototype on some well-known examples from the testing literature: the programs TriType, Bsort and Sample (see Appendix). Given the lengths of the sides of a triangle, Tritype carries out a series of tests on them to classify the triangle. It has no loops and only 14 execution paths but is interesting because the path predicates include simple arithmetical expressions and not just inequalities as in the other examples. Bsort is a bubble sort containing two nested loops, one iterating over all the elements of the array to be sorted and the other over the elements after the current one. This example demonstrates the limits of our current implementation of the k-paths strategy : the number of superfluous tests grows exponentially with k due to PathCrawler's attempts to find paths with k executions of both loops. The best way to limit the number of paths in this case is therefore by reducing the length of the array to be sorted. Sample compares the contents of two arrays to a reference value in two successive loops, each with a fixed number of iterations of the length of the array. For array lengths n and m, the number of paths is 1+ (2^n -1) * 2^m. The k-path strategy cannot be used for this example because the number of iterations is fixed but the number of paths can be kept reasonable by limiting n and m, which is justified by the total lack of dependence between successive loop iterations. The number of tests, number of infeasible prefixes, mean execution time in seconds and variation in the execution times over 10 runs are shown in Table 2

Table 2. Results for other examples

program	k	array dimn.	tests	infeasible prefixes	mean exec. time	min exec. time	max exec. time
TriType	-	-	14	3	0.01	0.01	0.02
Bsort	10000	0 - 5	153	349	1.16	1.14	1.17
Sample	-	4	241	0	0.27	0.22	0.29

9 Further Work

Our first priority is to apply PathCrawler to a wide range of larger examples. However, our results so far suggest that our approach is efficient enough to scale up to the treatment of larger programs providing that we limit the combinatorial explosion of the number of execution paths. We have shown how we easily adapted our test selection strategy to limit the number of iterations of certain

loops. Our current topics of investigation [11] include strategies to avoid testing all the paths in each call to another function. Our method is open to such modulations in the test 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. Integration test scenarios could be used in the same way. 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. By annotating the conditions in called functions, the exploration of different paths in these functions could be restricted.

However, the effectiveness of our test generation strategy is limited by the selection of a single test for each path. It could be easily modified to select tests at the limits of the path domain boundaries [4], where bugs are often found. The chances of detecting coincidental correctness would be improved if we also extended our random generation of variable values to the random generation of several tests for each path, in a similar way to statistical structural testing [16, 3].

Finally, the applicability of PathCrawler depends on a high degree of automation of the test process. In the current prototype, the oracle must be hand-coded but by taking certain forms of post-condition on the C variables into account, we could automatically generate the oracle as in [6, 13].

References

1. 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
2. A. Gotlieb, B. Botella and M. Reuher, A CLP Framework for Computing Structural Test Data, CL2000, LNAI 1891, Springer Verlag, July 2000, pp 399-413
3. 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
4. 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
5. B. Korel, Automated Software Test Data Generation, IEEE Transactions on Software Engineering, Vol. 16, No. 8, August 1990
6. 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 Formal Methods for Components and Objects, LNCS Vol. 2852, Springer Verlag, Berlin, 2003, pp 262-284
7. B. Marre and A. Arnould, Test sequences generation from Lustre descriptions: GATeL, ASE 2000, Grenoble, pp 229–237, Sep. 2000
8. B. Marre, P. Mouy and N. Williams, On-the-Fly Generation of K-Path Tests for C Functions, 19th IEEE Intnl. Conf. on Automated Software Engineering (ASE 2004), September 2004, Linz, Austria.
9. C. Michael and G. McGraw, Automated Software Test Data Generation for Complex Programs, ASE, Oct 1998, Honolulu

10. 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
11. P. Mouy, Vers une méthode de génération de tests boîte grise "à la volée", Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'04), June 2004, Besançon, France
12. G.C. Necula, S. McPeak, S.P. Rahul and W. Weimer, CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs, Proc. Conference on Compiler Construction, 2002.
13. M. Obayashi, H. Kubota, S.P. McCarron and L. Mallet, The Assertion Based Testing Tool for OOP: ADL2, In Proc. ICSE'98, Kyoto, Japan, 1998
14. R.E. Prather and J.P. Myers, The Path Prefix Testing Strategy, IEEE Transactions on Software Engineering, Vol. 13, No. 7, July 1987
15. N.T. Sy and Y. Deville, Consistency Techniques for Interprocedural Test Data Generation, ESEC/FSE'03, September 1-5, 2003, Helsinki, Finland
16. P.Thevenod-Fosse and H.Waeselynck, Software statistical testing based on structural and functional criteria, 11th International Software Quality Week, San Francisco (USA), 26-29 May 1998
17. M. Wallace, S. Novello and J. Schimpf, ECLiPSe: A Platform for Constraint Logic Programming, IC-Parc, Imperial College, London, August 1997

Appendix

```
int tritype(int i, int j, int k){
  int trityp;
  if ((i == 0) || (j == 0) || (k == 0)) trityp = 4;
  else {
    trityp = 0;
    if (i == j) trityp = trityp + 1;
    if (i == k) trityp = trityp + 2;
    if (j == k) trityp = trityp + 3;
    if (trityp == 0){
      if ((i+j <= k) || (j+k <= i) || (i+k <= j)) trityp = 4;
      else trityp = 1;
    }
    else if (trityp > 3) trityp = 3;
    else if ((trityp == 1) && (i+j > k)) trityp = 2;
    else if ((trityp == 2) && (i+k > j)) trityp = 2;
    else if ((trityp == 3) && (j+k > i)) trityp = 2;
    else trityp = 4;
  }
  return trityp;
}


void bsort (int * tableau, int l)
{
  int i, temp, nb;
  char fini;
  fini = 0;
  nb = 0;
  while ( !fini && (nb < l-1)){
    fini = 1;
    for (i=0 ; i<l-1 ; i++)
      if (tableau[i] < tableau[i+1]){
        fini = 0;
        temp = tableau[i];
        tableau[i] = tableau[i + 1];
        tableau[i + 1] = temp;
      }
```

```
    nb++;
  }
}


int sample(int a[4], int b[4], int target)
{
  int i, fa, fb;
  i=0;
  fa=0;
  fb=0;
  while(i<=3){
    if(a[i]==target) fa=1;
    ++i;
  };
  if(fa==1){
    i=0;
    fb=1;
    while(i<=3){
      if(b[i]!=target) fb=0;
      ++i;
    }
  }
  if(fb==1) return 0;
  else return 1;
}
```