Interleaving static and dynamic analyses to generate path tests for C functions

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 the PathCrawler method for the automatic generation of test-cases satisfying the all-feasible-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.

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. We describe a novel method for the efficient generation of tests for 100% coverage of feasible execution paths and show how it can easily be modified to satisfy the well-known *k*-path limitation on the number of loop iterations covered.

Related Work

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.

Most previous work on test-case generation for structural testing of sequential programs addresses the problem (called the Test Data Generation Problem (TDGP) in [5]) of finding

data to cover a test objective in the form of given node, branch or path of the control flow graph. 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 sketched out in [13], 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.

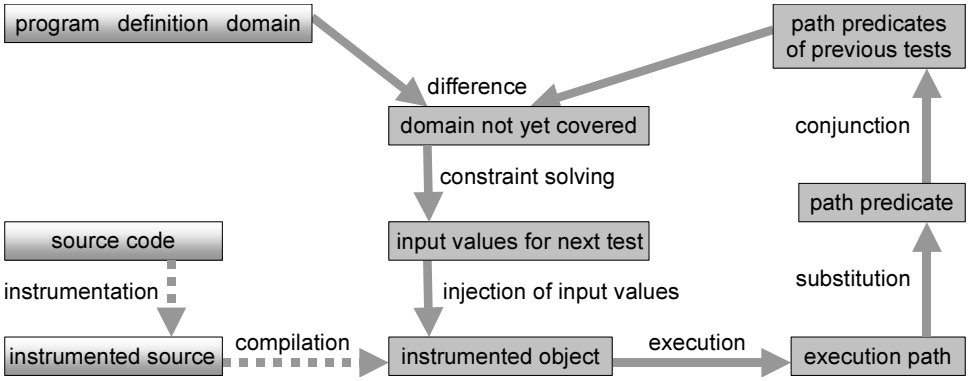
Like the dynamic approaches [1][5][9] to test-case generation, PathCrawler is based on dynamic analysis, but instead of heuristic function minimisation, it uses constraint logic programming to solve a (partial) path predicate and find the next test-case, as in the approaches based on static analysis [2][3][15]. It suffers neither from the approximations and complexity of static analysis, nor from the number of executions needed 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 a prototype has been implemented for C. This paper extends [8] and illustrates 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. This is the function `Merge`, whose source code is shown in Figure 1. `Merge` 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 PathCrawler to `Merge`, before concluding with a discussion of further work.

Figure 1 : source code of the function `Merge`

```
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
```

Figure 2 : our approach



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 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 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 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. 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.

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 implemented using the CIL library [12]

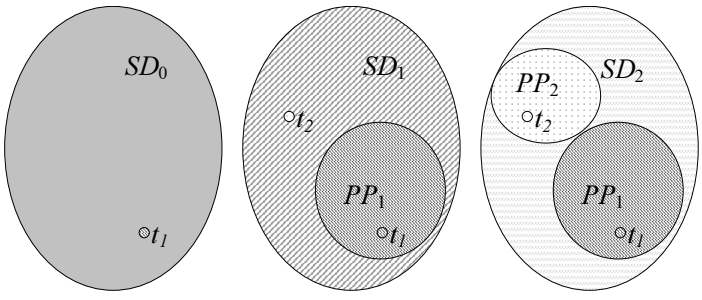
Substitution

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.

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 of SD_0 and the domain of PP_1, let us note SD_1 this selection domain (see Figure 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. To sum up this selection strategy, at each step *i*, we generate a test-case t_i from a selection domain SD_{i-1}, and when

Figure 3 : input domains



executed, this test exercises a path whose predicate is PP_i. Let us assume that there exist n feasible paths, then each SD_i can be recursively defined in the following way:

$SD_0 = ID$

$\forall\ i \in 1..n,\ SD_i = SD_{i-1} \wedge \neg PP_i = ID \wedge \neg PP_1 \ldots \wedge \neg PP_i$

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 :

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

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.

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 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-case, i.e. one covering a path which - after the prefix - executes more than k iterations of a loop.

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, *ID*, 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 dimensions which are variable 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. We also ask the user to define the input parameter dependencies (pre-condition). A limited form of universal quantification can be used in these definitions. In our example (see Figure 4), 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). Furthermore, the elements of `t1` must be ordered.

Test selection and constraint solving are implemented in the Eclipse constraint logic programming environment [16]. Solving, or determining the satisfiability of, a set of constraints on numbers with finite domains is decidable but NP-complete in the worst case but we use heuristics adapted from the constraint solving procedure used in [3] and [6] which display low complexity in practice. Our "labelling" heuristic (used to generate and test values after constraint propagation) is to choose data-structure dimension values as low as possible. For other values, labelling uses a random generator, biased towards the middle of the domain after constraint propagation. For constraints on floating-point numbers we currently use the incomplete procedure offered by Eclipse but specialised techniques to correctly handle arithmetic operations on floating-point numbers are currently being investigated in research [10][15] which holds the promise of decidable and precise constraint solving for these numbers too.

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.

Figure 4 : domains and pre-condition for `Merge`

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

and similarly for `t2`

Example

Now let us follow step by step what happens when we run PathCrawler on the example function `Merge`. In the first test-case of our example, generated from the domains and constraints of Figure 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 Figure 1), preceded by a sign indicating satisfaction or not 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 Figure 5:

$C_{1,1}$ = ¬ *Cond*3a: ¬ 0 < `l1` (exit 1st loop after 0 iterations)

$C_{1,2}$ = ¬ *Cond*11 : ¬ 0 < `l1` (exit 2nd loop after 0 iterations)

$C_{1,3}$ = ¬ *Cond*15: ¬ 0 < `l2` (exit 3rd loop after 0 iterations)

Solution of $MaxC_2$ = ¬*Cond*3a ∧ ¬*Cond*11 ∧ *Cond*15 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-case, 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. With no limit on loop iterations, $MaxC_4$ would be the conjunction of :

$C_{3,1}$ = ¬ *Cond*3a: ¬ 0 < `l1` (exit 1st loop after 0 iterations)

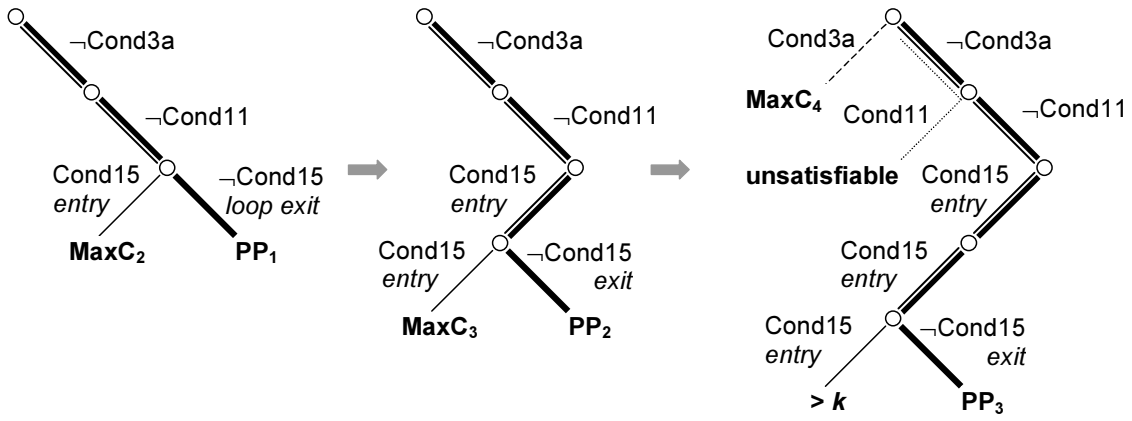
$C_{3,2}$ = ¬ *Cond*11: ¬ 0 < `l1` (exit 2nd loop after 0 iterations)

$C_{3,3}$ = *Cond*15: 0 < `l2` (entry 1st iteration of 3rd loop)

$C_{3,4}$ = *Cond*15: 1 < `l2` (entry 2nd iteration of 3rd loop)

¬$C_{3,5}$ = *Cond*15 : 2 < `l2` (entry 3rd iteration of 3rd loop)

Figure 5 : selection strategy



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 *k* iterations of the third loop. Our strategy thus backtracks to the lowest unexplored branch and constructs the path prefix $\neg$*Cond*3a $\wedge$ *Cond*11. However, this is unsatisfiable, so *MaxC*$_4$ is in fact *Cond*3a.

The 12th and 17th test-cases of our example contain more than 2 iterations of the same loop, although they are generated from prefixes (underlined in Table 1) containing only two iterations (occurrences of +3b). However, our strategy does enable us to prune the search space effectively after generation of a superfluous test. Pruning of infeasible paths is effective too: in this example, 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 the PathCrawler prototype ten times on `Merge` with *k* set to 5 and maximal domains for the elements of `t1` and `t2`. The control-flow graph then contains 4536 paths, of which 321 are feasible. In each run, 337 tests were generated and 317 infeasible path predicate prefixes, i.e. 654 predicate prefixes were generated and solved or rejected. The CPU execution times on a 2GHz PC running under Linux varied between 0.75 and 0.81 seconds, with an average of 0.785.

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							-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

Further work

Our first priority is to try out PathCrawler on other realistic program examples but the above results show that it promises to be an efficient method which makes the automatic generation of path tests for sequential programs feasible. The classic disadvantages of structural testing and the potential combinatorial explosion in the number of paths have discouraged the adoption of path testing but we believe that PathCrawler is sufficiently open to extension to be able to address these problems. We have already illustrated some ways in which our 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 pre-condition on the input values of the program under test. Other external constraints could be derived from a more complete specification of the 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. We are currently investigating [11] how PathCrawler can be extended in such ways to mixed functional/structural testing and to integration testing.

All-feasible-*k*-paths is a criteria which is rigorous enough to justify systematic, automated testing, with no reliance on human selection of test-cases (apart from fixing *k*), but its effectiveness is probably limited by the selection of a single test-case for each path. Our strategy is weaker on the detection of bad positioning of path domain boundaries than domain testing [4]. PathCrawler could be easily modified to select test-cases at the limits of the path domain boundaries, where bugs are often found. The chances of detecting coincidental correctness are lower than in statistical structural testing [3], but we could also modify our random generation of variable values to randomly generate several test-cases for each path.

The number of tests required for path testing means that the testing process must be made as automatic as possible. PathCrawler can integrate an oracle in the test generation and execution loop but it must currently be coded by the user. Any pre-condition on the inputs must be formalised by the user in order to run PathCrawler and by also taking certain forms of post-condition on the C variables (such as assertions in the code) into account, the oracle could be automatically generated, 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. Gottlieb, 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 et al., *How the Design of JML Accomodates both Runtime Assertion Checking and Formal Verifications*, In Formal Methods for Components and Objects, LNCS Vol. 2852, Springer Verlag, 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*, 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 2004), 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] M. Wallace, S. Novello and J. Schimpf, *ECLiPSe: A Platform for Constraint Logic Programming,* IC-Parc, Imperial College, London, August 1997