On-the-Fly Generation of K-Path Tests for C Functions

Nicky Williams, Bruno Marre, Patricia Mouy

On-the-Fly Generation of *K*-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 propose a novel method, called PathCrawler, for the automatic generation of structural tests satisfying the all-paths criterion or its k-*path variant. The source code is instrumented so as to recover the symbolic execution path each time that the program under test is executed. This code is first executed using inputs arbitrarily selected from the input domain. The resulting symbolic path is transformed into a path predicate by projection of the conditions onto the input variables. The next test is obtained by using constraint logic programming to find new input values outside the domain of the path which is already covered. The instrumented code is then executed on this test and so on, until all feasible paths have been covered. Our method combines static and dynamic analysis in a way that avoids the disadvantages of both. It is currently being implemented for the C language.*

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. 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.

This article describes PathCrawler, a novel method for the efficient generation of tests for 100% coverage of feasible execution paths. The strict interpretation of the all-paths criterion soon becomes unrealistic for programs containing loops with a variable number of iterations. 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*. We show that our method is easily modified to satisfy the *k*-path criterion.

2. Related Work

Most previous work on test data generation for structural testing of sequential programs addresses the problem 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 [2][4][10] 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.

Dynamic approaches [1][5][7] avoid the problems of symbolic execution by not using the path predicate. Instead, the program is instrumented so as to evaluate, at each execution, the "distance" from the test objective and general heuristic function minimisation techniques are used to search for input values to 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 and they may fail to find a test case even when one exists.

The contributions of this paper are the following:

1) We maintain that, 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 an idea for branch coverage sketched out in [9], but we do not leave feasible paths uncovered by limiting exploration of each previous path predicate to only one prefix.

2) If each path to be covered is selected from the control flow graph then the feasibility of each one must be checked. 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 path 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.

3. Our Approach

Our approach is illustrated in Figure 1. The source code is instrumented so as to recover the symbolic execution path each time that the program under test is executed. This code is first executed using inputs arbitrarily selected from the input domain. The path predicate is deduced from the resulting symbolic path. The next test is obtained by using constraint logic programming to find new input values outside the domain of the path which is already covered. The instrumented code is then executed on this test and so on, until all feasible paths have been covered.. This approach is currently being implemented for C.

4. Instrumentation

The instrumentation stage is an automatic transformation of the source code so as to print out the symbolic execution path, i.e. each assignment carried out and each condition satisfied during execution. The code is first transformed so as to eliminate conditional statements with side effects and multiple assignments, function calls or conditions in the same statement.

A trace instruction is then inserted after each assignment and each branch of the source code. All C data access paths are represented in a canonical form. To implement the *k*-path criterion, the trace of conditions in the head of loops is annotated with additional information used during constraint solving.

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, as well as other symbolic information needed in the case of aliases. When a condition is encountered, all occurrences of local variables are replaced by their current symbolic values. Because we analyse a single, unrolled, path, we

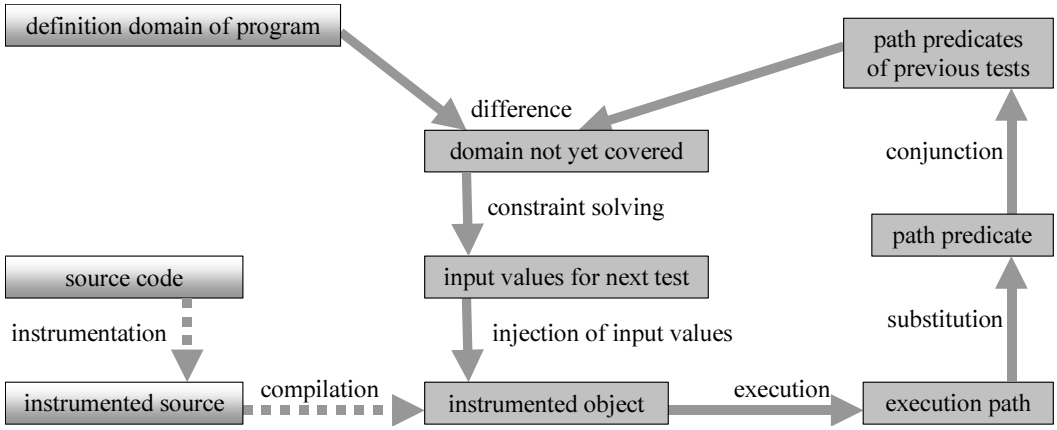

Figure 1 : our approach

do not need to use the SSA form used in [2] and instrumentation and alias analysis is greatly simplified. We can thus treat C data structures and pointers.

6. Test Selection and Constraint Solving

The first test case is obtained by picking input values inside the input domain DD_0 of the program under test. DD_0 is defined by a pre-condition consisting of the Cartesian product of the domains of the input parameters and a conjunction CD_0 of constraints reflecting input parameter dependencies. Domain definitions and constraints can be universally quantified to some extent.

The input values of the first test case, t_1, are found by solving of this constraint satisfaction problem. The consistency of the constraints to be solved throughout the test selection phase is decidable for (finite) C integer values and we use constraint logic programming techniques for computing their solutions. This can theoretically be NP-complete but we use heuristics developed for test case generation problems in [4][6] and which perform far better than this in practice. For floating-point numbers, we currently use the incomplete procedure provided by the Eclipse CLP environment [11], whilst awaiting the results of research [8][10] which holds the promise of decidable and precise constraint solving for these numbers too.

From the execution of t_1, we derive the corresponding path predicate PP_1. This 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. In order to cover a new path, we have to generate test inputs from the difference, DD_1, of DD_0 and the domain of PP_1 (see Figure 2). If DD_1 is not empty, we can generate a new test case t_2, from DD_1, which exercises a new path whose predicate is PP_2. This process is repeated until an empty selection domain DD_n is reached, in which case we have covered every feasible path of the program under test. Let us assume that there exist n feasible paths, then, given CD_0, each conjunction CD_i of constraints characterising domain DD_i can be recursively defined as follows:

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

$CD_i = CD_{i-1} \wedge \neg PP_i = CD_0 \wedge \neg PP_1 \wedge ... \wedge \neg PP_i$

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:

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

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} must contain $MaxC_i$ as prefix and so the negation of PP_{i+1} (expressed in the above form) subsumes the negation of all previous paths.

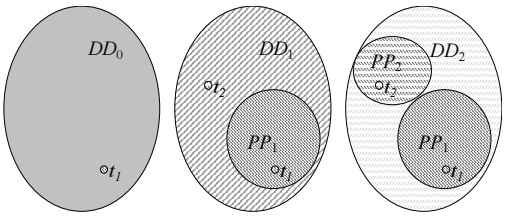

Figure 2 : input domains

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 will certainly give rise to paths containing more than k iterations of a loop then we do not explore it. 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. In the example shown in Figure 3, in which k is set to 2, the predicate PP_1 of the path covered by the first test t_1 contains the following conditions, of which the third is annotated: $C_1^1 = Cond_1$, $C_1^2 = Cond_2$, $C_1^3 = Cond_3$ (loop exit after 0 iterations).

From PP_1, we derive $MaxC_2 = Cond_1 \wedge Cond_2 \wedge \neg Cond_3$. Constraint solving of $MaxC_2$ generates the second test case in which there is one loop iteration. The third test effects two iterations and is generated in a similar way. With no limit on loop iterations, $MaxC_4$ would be the conjunction of :

$C_3^1 = Cond_1$, $C_3^2 = Cond_2$, $C_3^3 = \neg Cond_3$, $C_3^4 = \neg Cond_4$, $\neg C_3^5 = \neg Cond_5$ (entry 3rd loop iteration).

Because it would entail more than k iterations of the loop, this conjunction is not solved. Our strategy thus backtracks to the lowest unexplored branch and constructs the path prefix $Cond_1 \wedge \neg Cond_2$. Suppose, however, that this is unsatisfiable. $MaxC_4$ is then $\neg Cond_1$.

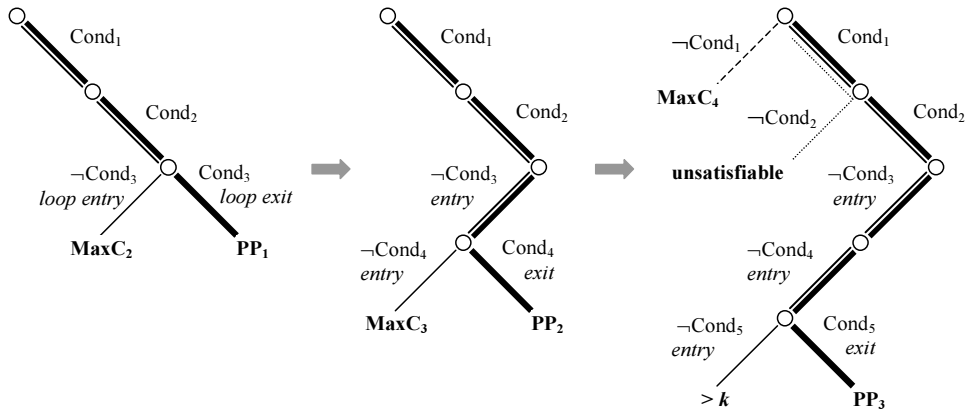

Figure 3 : an example of the test selection strategy

We tried out our prototype implementation on an example program containing many infeasible paths (see [12]). *k* was set to 5, giving 4536 theoretical paths. 337 tests were generated, of which 16 contained more than 5 loop iterations, and 317 infeasible path predicate prefixes were detected to eliminate the 4215 infeasible paths. The CPU time, excluding execution of the 337 tests, was 1.6 seconds on a 2GHz PC running under Linux.

7. Further work

Our basic test generation method provides a good starting-point for the design and automation of test selection strategies which alleviate the disadvantages of pure structural testing. This is because it is both efficient and open to various types of modification, as we have shown.

Firstly, information collected during execution of the program under test can be used to influence test selection, as illustrated by the use of annotations of loop-head conditions to implement the *k*-path criterion.

Secondly, 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.

If we had not only a pre-condition but a more complete specification of the program under test in the form of pre- and post-conditions on the C variables (or assertions at the program entry and exit points), then we could implement a grey-box testing strategy. This would consist of analysing the specification to define functional domains, each of which would then be covered structurally. Such a specification would also enable us to automatically generate the oracle, which must currently be hand-coded.

We could also use constraints derived from such a specification to replace function calls and implement a structural integration testing strategy.

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] 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

[4] 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

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

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

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

[8] 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

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

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

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

[12] N. Williams, B. Marre and P. Mouy, *On-the-fly Generation of* K*-Path Tests for C Functions*, Rapport DRT/LIST/DTSI/SOL/LSL/04-162, CEA, France, 2004