A New Way of Automating Statistical Testing Methods

S.-D. Gouraud[1], A. Denise[1], M.-C. Gaudel[1], B. Marre[1,2]

[1]L.R.I., Université Paris-Sud, bât. 490
91405 Orsay Cedex, France.
{gouraud, denise, mcg, marre}@lri.fr

[2]CEA, LIST/DTSI/SLA/LSL
91191 Gif sur Yvette Cedex, France.
Bruno.Marre@cea.fr

Abstract

We propose a new way of automating statistical structural testing, based on the combination of uniform generation of combinatorial structures, and of randomized constraint solving techniques. More precisely, we show how to draw test cases which balance the coverage of program structures according to structural testing criteria. The control flow graph is formalized as a combinatorial structure specification. This provides a way of uniformly drawing execution paths which have suitable properties. Once a path has been drawn, the predicate characterizing those inputs which lead to its execution is solved using a constraint solving library. The constraint solver is enriched with powerful heuristics in order to deal with resolution failures and random choice strategies.

Keywords: *software testing, combinatorial structures, constraint solving, statistical and random testing, structural testing*

1. Introduction

Statistical or random testing methods are based on a probability distribution on the input domain of a piece of software. This distribution is used to draw arbitrary number of test data, allowing intensive test campaigns. Various kinds of distributions have been studied: Uniform distributions [1, 5], operational distributions derived from the operational profile of the environment of the future system [12], or distributions which are derived from the structure of the program [15] or of the specification [16].

This last class of methods is based on the construction of a probability distribution on the input domain which, given a set of elements to cover w. r. t. some coverage criteria, maximizes the weakest probability for an element to be activated by an execution. For instance, structural statistical testing based on the all-statements coverage criterion leads to the construction of an input distribution which avoid too weak probability for a statement to be exercised, even if this statement is only used for a small subset of the input domain. Similarly, all-branches, all-paths, or any other structural coverage criteria can be used as a basis for structural statistical testing. Functional criteria, i.e. coverage criteria based on the specification, can be used as well. These methods provide a way to combine random testing and coverage requirements, such that no element in the set to be covered being seldom or never exercised during testing.

In this paper we present a new way of automating structural statistical testing using the CS package for counting and randomly generating combinatorial structures [4]. This tool is part of the MuPAD system [14]. Uniform generation of so-called labeled combinatorial structures has been studied in the context of the average case complexity analysis of algorithms [7]: this analysis requires counting all the possible input cases of a given size and considering a uniform probability distribution on them. It turns out that execution paths in control flow graphs can be modeled by such structures. Thus CS provides a way of drawing uniformly paths of a given length of a program. Moreover, we show how to use it for other sets of paths, for instance those of length less than a given bound, those passing through a specific node or edge of the graph. It is worth noting that this approach is applicable to any test criteria based on any graph or tree. Once a path has been drawn, the predicate characterizing those inputs which lead to its execution is built using classical symbolic evaluation techniques [18]. Then, it is solved using a constraint solving library based on the ECL^{i}PS^e system [6] enriched with powerful heuristics in order to deal with resolution failures and random choice strategies. This library is adapted from the one developed and exploited in the GATEL system for LUSTRE programs [11].

Thus we propose a new way of automating structural statistical testing, based on the combination of uniform generation of combinatorial structures and of randomized constraint resolution techniques. This approach is significantly different from the one described in [15] since it is based on the generation of execution paths, not on the construction of a distribution on the input domain.

The paper is organized as follows. Section 2 introduces the principles of uniform random generation of paths in a directed graph. In Section 3, we present our method for the All paths criterion, test data generation and the principles for our process. Section 4 extends our method to the All statements and All branches criteria. Related works are discussed in Section 5. Finally, Section 6 concludes the paper and suggests some future work.

2. Uniform random generation of paths in a directed graph

As outlined above, a first step of our methodology consists in generating random execution paths in the control flow graph of the program that we deal with. In the present section, we consider the control flow graph as an (ordinary) connected directed graph where two vertices v_b and v_e are distinguished as the beginning and end points respectively. The problem is the following : given an integer n, we have to generate at random one or several paths of length $\leq n$ from v_b to v_e, in such a way that all possible paths have the same probability to be generated. (The length of a path is the number of edges that it crosses.)

At first, let us focus on a slightly different problem: the generation of paths of length n exactly. We will see further that a small change in the graph will allow to generate paths of length $\leq n$. The problem of generating paths of a given length in a graph is equivalent to the one of uniform random generation of words of *regular languages*. Indeed, a regular language is defined by a particular graph called *finite state automaton*, and any word of the language corresponds to a path in the automaton. The problem of generating words of regular languages has first been discussed in [9], and the method has been improved and widely generalized in [7]. The principle of the generation process is simple: Starting from vertex v_b, one draws a path step by step; at each step, the process consists in choosing a successor of the current vertex and going to it. The problem is to proceed in such a way that only (and all) paths of length n can be generated, and that they are equiprobably distributed. This is done by choosing successors with suitable probabilities. Given any vertex v of the graph, let $f_m(v)$ denote the number of paths of length m which connect v to the end vertex v_e. Suppose that, at any step of the generation, we are on vertex v which has k successors denoted $v_1, v_2, \ldots, v_k$. In addition, suppose that $m > 0$ edges remain to be crossed in order to get a path of length n. Then the condition for uniformity is that the probability of choosing vertex v_i $(1 \leq i \leq k)$ equals $f_{m-1}(v_i)/f_m(v)$. In other words, the probability to go to any successor of v must be proportional to the number of paths of suitable length from this successor to v_e.

So there is a need to compute the numbers $f_i(v)$ for any $0 \leq i \leq n$ and any vertex v of the graph. This can be done by using the following recurrence rules:

$$\begin{array}{rcll} f_0(v) & = & 1 & \text{if } v = v_e \\ & = & 0 & \text{otherwise} \\ f_i(v) & = & \sum_{v \to v'} f_{i-1}(v') & \text{for } i > 0 \end{array}$$

where $v \to v'$ means that there exists an edge from v to v'.

Now the generation scheme is as follows:

- Preprocessing stage: Compute a table of the $f_i(v)$'s for all $0 \leq i \leq n$ and all vertices.
- Generation stage: Draw the path according to the scheme seen above.

Note that the preprocessing stage must be done only once, whatever the number of paths to be generated. Easy computations show that the memory space requirement is $n \times |G|$ integer numbers, where $|G|$ stands for the number of vertices in the graph. The number of arithmetic operations needed for the preprocessing stage, as well as for the generation stage, is linear in n.

Now we address the problem of generating paths of length $\leq n$ instead of exactly n. The only change is the following: Add to the graph a loop edge from v_b to itself. Each path of length n from v_b to v_e in this new graph crosses k times the new edge, for some k such that $0 \leq k \leq n$. With this path we associate a path of length $n - k$ in the previous graph, by removing the k loop edges. It is straightforward to verify that any path of length $\leq n$ can be generated in such a way, and the generation is uniform.

In our implementation, the generation of paths is programmed in MuPAD, using the CS package. MuPAD [14] is a formal and algebraic calculus tool, developed in University of Paderborn. CS [3, 4] is a package devoted to counting and randomly generating combinatorial structures, based on the general notion of "decomposable structures" defined in [7]. Thus the formalism used in our implementation of the algorithm of generation of paths is different, while similar, to the simple version described above. But the principles –and the results– are unchanged.

3. First application: All paths criterion

In this section, we show how to use combinatorial structures to represent control flow graphs and how we use CS for paths generation. Then we recall the principles of construction of path predicates and present the predicate resolution method used to obtain test data. The actual coverage criterion implemented here is "all the control flow paths of length less or equal to n". We will see later that combining this criterion with statistical testing is not very interesting (as already noticed in [15]), but this presentation is the basis for the presentation of other structural criteria in section 4.

We use a simple imperative programming language inspired from C and Pascal. The basic constructions are:

- sequential composition
- *If...Then...Else* construction (*Else* is optional)
- *While* loop
- *For* loop which becomes a sequential composition (when iterations are fixed) or a *While* loop

The data types are booleans and integers.

3.1. Control flow paths generation

Classically, a program is associated with its control flow graph: The nodes are maximum indivisible blocks of statements, or predicates which appear in the conditional statements, and the edges correspond to the possible transfers of control between these nodes. Two nodes are distinguished as the *Init* and *Exit* nodes; they correspond to the beginning and the end point of the program. Each node (resp. edge) is labeled in order to find easily at which piece of code (resp. branch) of the program it corresponds to. A *control flow path* is a path in the control flow graph which goes from *Init* to *Exit*.

For using CS, we need to describe the control flow graph by a combinatorial specification, as defined in [7]. A combinatorial specification of a given class of combinatorial structures consists in a set of production rules made from basic objects (called *atoms*), and from operators on classes and atoms. Among the six possible operators, the specification of a control flow graph needs only two: *Union* and *Prod*.

Example 1 (Merge: algorithm)
input*: t1, t2: array of int; IMax, JMax:int*
output*: t3 array of int*
vars*: i, j, k: int*
begin
i:=1; j:=1; k:=1;
while*((i≤IMax) and (j≤JMax))*{
if*(t1[i]<t2[j])*
then{*t3[k]:=t1[i];i:=i+1*}
else{*t3[k]:=t2[j];j:=j+1*}*;*
k:=k+1}*;*
while*(i≤IMax)*{
t3[k]:=t1[i];
i:=i+1; k:=k+1}*;*
while*(j≤JMax)*{
t3[k]:=t2[j];
j:=j+1; k:=k+1}*;*
end

Here is a specification corresponding to the Merge program:
specMerge:={*a0= Atom, i0= Atom, b1= Atom, s1= Atom, e2= Atom, t2= Atom, i3= Atom, i4= Atom, i5= Atom, b6= Atom, s6= Atom, i7= Atom, b8= Atom, s8= Atom, i9= Atom, Init= Prod(a0,I0), I0=d(i0,B1), B1= Union(Prod(s1,B6),Prod(b1,Prod(C2,B1))), C2= Prod(Union(Prod(t2,i3),Prod(e2,i4),i5)), B6= Union(Prod(s6,B8),Prod(b6,Prod(i7,B6))), B8= Union(s8,Prod(b8,Prod(i9,B8)))*}

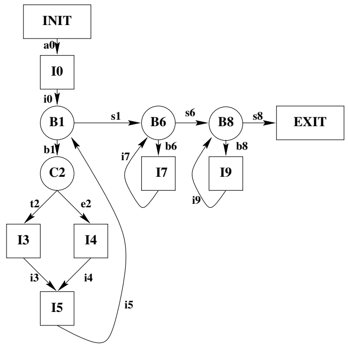


Figure 1. Control flow graph for Merge

We consider the nodes of a control flow graph as being classes, and the edges as being atoms. Each class refers to the label of a node, and represents the whole set of paths beginning at this node and ending at the *exit* node of the control flow graph. In the specification above, the *Init* class denotes the whole set of control flow paths.

The operator *Prod* stands for concatenation of paths. Thus the production *Init= Prod(a0,I0)* means that a path beginning at node *Init* is made of a first step labeled *a0*, followed by a path beginning at *I0*.

The operator *Union* stands for disjoint union. So, *B6= Union(Prod(s6,B8),Prod(b6,Prod(i7,B6)))* means that a path beginning at node *B6* consists either in the concatenation of edge *s6* and a path beginning in *B8*, or the concatenation of the two edges *b6* and *i7* with another path beginning at *B6*. A compiler performing the translation of a program into a combinatorial specification is described in [8].

As said above, we are interested in drawing paths with length less than or equal to n. This requires a minor transformation of the CS specification: We add an atom, named *virtual*, corresponding to the loop-edge mentioned in Section 2. Thus the specification of the Merge program becomes:
specMerge:={*...,*
virtual=Atom,
Init= Union(Prod(virtual,Init), Prod(a0,I0)),
...}

Given such a specification and an integer n, CS makes it possible to generate uniformly at random any number of control flow paths of length $\leq n$.

Let us call a *minimal path* a path of the graph such as each edge is crossed only once. If n is too small, there is a risk not to have all the minimal paths of the graph, even no path. Thus, n must be at least the length of the longest minimal path in the control flow graph.

The bigger is n, the closer we are to the All paths criterion. But in presence of loops, the size of the set of paths of length less or equal to n grows exponentially even for small programs. A lot of tests should be carried out to obtain a satisfactory coverage with a good probability [15]. Consequently, it is necessary to study weaker criteria such as All statements or All branches (cf. Section 4).

3.2. Test data generation

Given a set of control flow paths, we have to find valuations of inputs such that these paths are followed during program execution. Test data generation is done by resolution of the predicates characterizing the inputs causing the execution of a given control flow path.

3.2.1. From control flow paths to path predicates

Let us call *an execution path* in a program the sequence of statements and conditions occurring in the elements of some control flow path. These statements and conditions can be either assignment statements or boolean expressions from one of the outgoing edges of a conditional or loop statement. (A condition is negated when it corresponds to the `Else` branch or to the exit of a `IfThen` or a `While` statement).

A *path predicate* is the condition on inputs which ensures the execution of the corresponding execution path. Classically, a path predicate is built by traversing the execution path in the following way:

- the initial value of each variable x is denoted by x_0 ;
- each assignment statement $x := expr$ is transformed into an equality $x_{i+1} = \widetilde{expr}$ where i is the index of the current value of x and $\widetilde{expr}$ is the expression $expr$ where each variable occurrence has been replaced by the denotation of its current value. The current value of x is now denoted by x_{i+1}.
- each condition C, coming from a conditional or a loop statement, is replaced by a predicate $\tilde{C}$ where each variable occurrence has been replaced by the denotation of its current value.

The resulting path predicate is the conjunction of all these formulae.

Moreover, it is possible to take into account some preconditions on input variables. These preconditions are dealt with in the same way as the conditions coming from conditional or loop statements. Currently, we consider only comparisons and definitions of ranges for integer variables, but further extensions are planned.

Example 2 *A path predicate from the Merge program (see example 1).*
The control flow path a0 i0 b1 t2 i3 i5 s1 s6 b8 i9 s8 *corresponds to the following path predicate (each item gives the formula corresponding to each pair of consecutive edges labels) :*

a0-i0	$i_1 = 1 \wedge j_1 = 1 \wedge k_1 = 1$
i0-b1	$\wedge\ (i_1 \leq IMax_0)\, and\, (j_1 \leq JMax_0)$
b1-t2	$\wedge\ t1_0[i_1] < t2_0[j_1]$
t2-i3	$\wedge\ t3_1[k_1] = t1_0[i_1] \wedge i_2 = i_1 + 1$
i3-i5	$\wedge\ k_2 = k_1 + 1$
i5-s1	$\wedge\ not((i_2 \leq IMax_0)\, and\, (j_1 \leq JMax_0))$
s1-s6	$\wedge\ not(i_2 \leq IMax_0)$
s6-b8	$\wedge\ j_1 \leq JMax_0$
b8-i9	$\wedge\ t3_2[k_2] = t2_0[j_1] \wedge j_2 = j_1 + 1 \wedge k_3 = k_2 + 1$
i9-s8	$\wedge\ not(j_2 \leq JMax_0)$

`IMax` *and* `JMax` *cannot have negative values (the input arrays* `t1` *and* `t2` *contain at least one element), thus we can add the following preconditions :*

$$IMax_0 > 0 \wedge JMax_0 > 0$$

Once path predicates are built, it remains to solve them in order to obtain the actual test data, i.e. input values ensuring the execution of the corresponding paths.

3.2.2. Path predicate resolution. Path predicate resolution is done through a constraint logic programming interpretation of the predicates. As seen above, these predicates are conjunctions of boolean expressions. Each boolean expression is translated into logical constraints and a constraint solving package is used to compute solutions of the resulting constraint system. This package is extracted from the GATeL tool [11] which automates test sequences generation from LUSTRE descriptions (LUSTRE is a synchronous data flow language used for the description of reactive systems). The experimental results obtained on industrial cases studies show that this kind of constraint logic programming interpretation can successfully handle “real size” problems [11].

The constraint solver is built over finite domain libraries and coroutining mechanisms provided by the ECL^{i}PS^e environment [6]. Boolean variables take values inside the boolean domain, boolean operations are handled by symbolic simplifications and unification, integer variables take values inside union of integer intervals (bounded by “minInt” and “maxInt”), and arithmetic operations and comparisons are handled by reductions of interval domains and unification.

3.2.3. Constraint solving. As usual in constraint logic programming, the constraints are sequentially introduced and individually managed in a *constraint store*. A constraint propagation mechanism tries to simplify them as much as possible through integer interval reductions and symbolic simplifications. When a constraint cannot be further simplified, it “falls asleep”, waiting inside the constraint store for the instantiation of some variables that could make it progress further. Resolution proceeds by successive elimination of all constraints. An instantiation procedure, so-called *labeling* in the logic programming community, instantiates the variables involved in the constraints. A variable is instantiated inside its domain. If this valuation leads to a resolution failure, resolution *backtracks* to the other possible values.

Let us remark that usual labeling procedures are not random, while in our case variables are *randomly instantiated*. The valuations are tried according to a uniform distribution inside the known domain of a variable. By opposition to classical labeling strategies, our random strategy does not give preference to any value inside a domain.

In order to avoid erroneous valuations, the propagation mechanism continuously checks constraints satisfiability. The instantiation of a variable "awakes" the propagation of related constraints, which can disappear (when solved), or awake/create other constraints or lead to a resolution failure (empty domain or unification failure).

The equation defining a variable is introduced as a constraint only when needed, i.e. when some constraint needs the value of this variable. This "lazy" constraint propagation makes it possible to minimize the average number of constraints and then, the amount of memory needed (memory also depends of the number of "labeling" steps).

The efficiency of any constraint solver relies a lot on the choice of the variable to be instantiated during the labeling steps. Classical choice heuristics aim at awaking the maximal number of constraints, while minimizing the average branching in the resolution tree (choice among the variables with the smallest domain). In our choice heuristic we add two preliminary criteria. Among the variables concerned by some waiting constraints, we choose:

1. a variable which does not depend functionally on another variable (this criterion gives priority to initial values and corresponds to a "by value" evaluation strategy),
2. with the smallest index (in order to reflect the sequential evaluation),
3. with the smallest domain (in order to minimize the average branching in the resolution tree),
4. waking the maximum number of constraints.

The major benefit coming from the added criteria is that the resulting heuristic awakes indirectly more constraints than the classical ones. This comes from the observation that the propagation mechanism propagates more informations from the arguments of constraints than from their result. For example, an addition constraint can do more interval reductions when one of its arguments is instantiated than in the case where its result is instantiated.

When all constraints are solved, we get a partial instantiation of inputs (some input values were not needed during resolution). Tests are then computed by a random instantiation of the remaining input variables (inside their interval domain for integer variables).

3.2.4. Unfeasible paths. The existence of infeasible paths is an habitual problem for all structural testing methods [2]. There is no general algorithm allowing to identify them since this problem is known to be undecidable [18]. Things may seem easier in our case since the number of constraints and the domains of variables are finite. However, the time spent for the detection of an unsatisfiability may be discouraging (the size of the resolution tree may be impracticable).

In most cases, such unsatisfiabilities are fast detected by the constraint propagation mechanism. Otherwise, unsatisfiabilities are detected consecutively to a labeling step (see Section 3.2.3) which may lead to a very slow enumeration of the domain of a variable (due to the propagation of many constraints and the backtracks made for each possible value until reaching an empty domain). In order to shorten the response time of the solver, we have chosen to limit the time allowed to solve a path predicate (10 seconds). When a resolution "time-out" is reached, the path predicate resolution is retried from the beginning (without backtracking). This can be done up to a maximum number of attempts (10 in the current implementation). Since the valuations are randomly chosen, the probability to make the same consecutive choices from one resolution attempt to another is very small. Furthermore, the random value chosen during a labeling step may lead to very different propagations which could reduce very differently the domain of some variables. Thus, the variable choice heuristic may choose different variables from one attempt to another. In most cases, this "multiple attempts" strategy is far more efficient to find solutions than the classical backtracking strategy (especially when many consecutive choices where made inside "big" integer domains).

When the maximum number of attempts has been reached without finding a solution, the predicate resolution is aborted. This leads to a *theoretical* loss of the completeness property: it is unknown whether the corresponding path is feasible or not. In fact, the practical cases where our bounded resolution is not complete corresponds to the cases where the response time would be unacceptable with a classical backtracking strategy.

However, we are able to distinguish causes of resolution failures according to the fact that an abort occurred or not (the maximum attempts number has or has not been reached). A failure that is not due to an abort corresponds to an infeasible path. This is an important piece of information for the tester, and it can be used to prune the set of selectable paths (this point will be discussed in conclusion). Besides, a failure due to an abort may also provide meaningful information for the tool developer. An accurate analysis could figure out improvements of the propagation mechanism/labeling heuristic.

4. All branches and All statements criteria

In the case of All statements or All branches criteria, the objective is to draw random execution paths in a way such that all statements (resp. branches) have balanced probabilities to be covered. This can be stated more precisely by using the notion of *test quality* defined in [15]. Let us call *element* a block of statements (All statements criterion), or a branch (All branches criterion). The quality of a test of N executions, denoted q_N, equals the minimal probability for any element to be exercised during the N executions. Let E be the set of elements (branches or statements, according to the criterion), and let $p(e)$ denote the probability for any element $e \in E$ to be exercised during *one* execution. Then we have

$$q_N = 1 - (1 - p_{\min})^N$$

where $p_{\min} = min\{p(e), e \in E\}$.
Thus, for any given N, in order to improve the test quality q_N, it is necessary to maximize $p_{\min}$.

Let us denote C_n the set of execution paths of length $\leq n$ and, given a statement or a branch e, let C_n^e be the set of paths with length $\leq n$ which pass through e. Our approach for maximizing $p_{\min}$ consists in selecting a suitable subset $S \subset E$ and following N times the scheme below (the choice of S will be detailed later):

1. Draw randomly an element e from the set S.
2. Draw randomly one path in C_n^e.

Following this scheme, let us compute $p(e)$ for any $e \in E$: The probability to draw randomly an element $e \in E$ in step 1 equals $\frac{1}{|S|}$ if $e \in S$, zero otherwise; and the probability of reaching e by drawing a random path for another element $e' \in S$ is $\frac{|C_n^{e,e'}|}{|C_n^{e'}|}$ where $C_n^{e,e'}$ denotes be the set of paths with length $\leq n$ which pass through e *and* trough e'. Thus we have

$$p(e) = \frac{[e \in S]}{\mid S \mid} + \frac{1}{\mid S \mid} \sum_{e' \in S \setminus \{e\}} \frac{|C_n^{e,e'}|}{|C_n^{e'}|} \quad (1)$$

where $[e \in S]$ equals 1 if $e \in S$, zero otherwise.

Note that the second step above requires generating control flow paths which are subject to an additional constraint: The paths must cross a given element (edge or node) of the control flow graph. The simplest way to perform this is to construct a new graph which represents the control flow graph subject to the new constraint. This can be done by considering the set of control flow paths as a regular language and the control flow path as its automaton. Then the new "constrained" graph is made by intersecting this language with the whole (regular) language of words which contain at least one occurrence of the given element. A standard algorithm for constructing automata of intersecting regular languages is given in [10].

Now the set S must be chosen in such a way that any path of C_n can be generated. Regarding $p(e)$, notice that some elements of E have $p(e) = 1$, even if they do not belong to S. This is the case for I0, B1, B6 and B8 in Figure 1. Remark also that some elements are correlated: For instance, any path which goes through C2 goes also through I5. On the other hand, equation (1) suggests that, for increasing $p_{\min}$, the cardinality of S is worth being minimal.

Let us illustrate these remarks with a new example.

Example 3 (All statements criterion)
In figure 2, let $S = E = \{C1, C2, I3, I4, I5\}$. We have $p_{\min} = p(I5) = \frac{1}{5} + \frac{1}{5} * \frac{1}{3} = \frac{4}{15}$.

Choosing $S = \{I3, I4, I5\}$ is a better solution: We get $p_{\min} = p(I3) = \frac{1}{3}$. Dropping $C1$ and $C2$ from S improves $p_{\min}$ since $p(C1) = 1$ in any case, and $C2$ is correlated with $I3$ and $I4$.

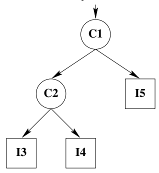


Figure 2.

Before presenting our general heuristic for the choice of S, we need some definitions. Let us suppose that the portion of code which corresponds to the body of a *While* loop or to the body of a *Then* (resp. *Else*) branch is delimited by *Begin* and *End*. Then we call *conditional subgraph*, the control flow graph which correspond to the code between a *Begin* and the corresponding *End*.

Example 4 *In the graph of the Merge program (fig. 1), there are five conditional subgraphs of which:*
(1) the subgraph resulting from the node $B1$ contains edges $b1$, $t2$, $i3$, $e2$, $i4$ and $i5$, and nodes $C2$, $I3$, $I4$ and $I5$
(2) the subgraph resulting from the node $C2$ contains edges $t2$ and $i3$, and node $I3$
(3) the subgraph resulting from the node $B6$ contains edges $b6$ and $i7$, and node $I7$

Definition 1
An element is mandatory *if all paths of the graph pass through this element.*
A element is internally mandatory *if it belongs to a conditional subgraph of the graph, containing at least two blocks, such that in this conditional subgraph it is an mandatory element.*
An independent *element is an element which is neither mandatory nor internally mandatory.*

Example 5 *In the graph of the Merge program (fig. 1): The mandatory elements are: $I0$, $B1$, $B6$, $B8$, $a0$, $i0$, $s1$, $s6$ and $s8$*
The internally mandatory elements are: $C2$, $I5$, $b1$ and $i5$
The independent elements are: $I3$, $I4$, $I7$, $I9$, $t2$, $e2$, $i3$

Now our heuristic for constructing the set S is based on the following scheme:

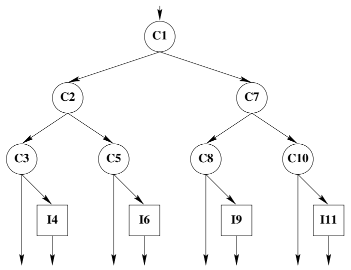
1. Initialize S with all independent elements.
2. Complete S until all paths can be covered.

Other possible strategies are presented in [8]. There exist various ways for performing the completion phase (phase 2). This phase occurs if there is at least a path which does not pass through any independent element.

Example 6 *In example 1, initially S is $\{I3, I4, I7, I9\}$. The path* `a0 i0 s1 s6 s8` *does not have any chance to be drawn. Thus we need to add a block among $\{I0, B1, B6, B8\}$ to allow this path to be drawn.*

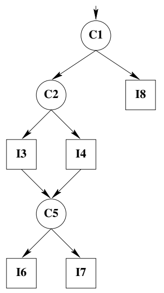
A simple heuristic for the completion is to add some mandatory element. The advantage of this choice is that the cardinality of S increases only by 1, while the coverage of all the paths of the graph is ensured.

Example 7 *Here, the choice of $C1$ to supplement the set $S = \{I4, I6, I9, I11\}$ is the best possible one. Indeed, the optimal solution is $P_{S\cup\{C1\}} = p(I4) = \frac{1}{5} + \frac{1}{5} * \frac{1}{8} = \frac{9}{40}$*



This construction of S looks attractive. Unfortunately there are control flow graphs for which it is not the best solution.

Example 8 *Here, our method gives us: $S = \{I3, I4, I6, I7, I8\}$ and thus $P_S = p(I8) = \frac{1}{5}$. But if we take the set $S' = \{I3, I4, I8\}$, then $P_{S'} = p(I3) = \frac{1}{3}$. S' is one of the optimal solutions sets.*



However, this heuristic gives an optimal result for a broad class of programs and it has the advantage of efficiency. We implemented it in our prototype while seeking if there is a better solution, or an optimal solution.

5. Related work

The first proposal for statistical structural testing comes from P. Thévenod-Fosse. After studying the advantages and the drawbacks of structural testing and of statistical testing, P. Thévenod-Fosse and H. Waeselynck have proposed a method combining statistical techniques and structural criterion. Their method consists in the construction of a new distribution of the input domain enabling to balance the probability to cover the less frequent elements. This approach has good experimental results [17, 15] but is not completely automated. In the approach presented here, we avoid the construction of the input distribution by drawing uniformly among subsets of execution paths. With this point of view, we can use the CS tool and have an automated statistical structural testing.

The idea to use constraint solving techniques for test data generation seems quite natural. However, there exists few references describing its use either for black-box testing [13] or for structural testing automation. In [2], a constraint logic programming interpretation is also used for the automation of test data generation (for the “All statements” and “All branches” structural criteria). The main differences with our approach concern essentially the nature of the translation and the solving heuristics used. In their approach a constraint logic program is generated. Its control flow graph contains all the control flow paths in the “slices” reaching each statement/branch of the original control flow graph. When invoked with a given statement/branch identifier, this logic program can dynamically build and solve any path predicates reaching this identifier. Constraints resolution uses constraint propagation like in our approach, but this is done by applying (costly) constraint entailment checks, and the labeling strategy seems to be a deterministic iterative domain splitting strategy combined with a classical heuristic for the choice of variables. Some experiments comparing this strategy to our four level heuristic should be done (see section 3.2.3). Like in our approach, this work have the problem of infeasible paths. A “time-out” mechanism is also used but only one resolution attempt is made. Our experiments have shown that the combination “random labeling/multiple attempts” is more efficient to solve complex constraint systems than deterministic strategies.

6. Conclusion and Perspectives

To our knowledge, this is the first attempt of combining random generation of combinatorial structures and constraint resolution in order to perform automatic statistical structural testing. On this basis, we have developed a prototype which, given a program under test, a coverage criterion and a size N, returns N test data. Several classical small programs (such as the Merge example) have been successfully processed. the next step consists in an experimental comparison of our approach with the one of Thévenod-Fosse and Waeselynck [15]: we are reusing some of their programs and mutants to evaluate the fault detection capability of the test sets produced by our tool. The results should be similar. The main difference is that we avoid the construction of a distribution on the input domain: We use existing tools to draw uniformly among subsets of execution paths. But the principle of the method, i. e. maximizing the probability of coverage of the less frequent elements, remains the same.

The possibility of drawing infeasible paths in the generation stage (see paragraph 3.2.4) is a problem in our approach. However, the combination of some possibilities of the constraint solver and CS could help: The infeasible paths detected during the introduction of the constraint system in the constraint store (see paragraphs 3.2.3 and 3.2.4) can be used to prune the set of selectable paths. Our constraint resolution method allows us to know at which point in the control flow path a failure occurs. This means that we are able to build a *control flow path prefix* representative of some/many infeasible paths. These prefixes could be used to modify the combinatorial specification of the set of selectable paths, or stored in an efficient tree structure in order to reject some paths during path selections.

Finally, other possibilities of CS for statistical testing are worth being explored: Some additional properties of the programs to be tested, which are not explicit in a control flow graph, can be taken into account in a combinatorial specification. More generally, other models than control flow graph may give rise to a combinatorial specification either based on the program or on a specification (i.e. statistical functional testing). Furthermore, CS could be used in a different approach for statistical testing, e.g. for drawing random data when the entry domain is complex (trees, graphs…)

7. Acknowledgments

We thank Sylvie Corteel for fruitful theoretical discussions at the beginning of the work. We also are grateful to Pascale Thévenod-Fosse for the explanations, examples and data that she has kindly provided.

This work is partially supported by the European community (IST Project 1999-11585: DSoS)

References

[1] D. Bird and C. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(2):229–245, 1983.

[2] B. Botella, A. Gotlieb, and M. Rueher. A CLP framework for computing structural test data. *First Internationnal Conference on Computational Logic CL 2000*, july 2000.

[3] S. Corteel, A. Denise, I. Dutour, F. Sarron, and P. Zimmermann. CS web page. http://dept-info.labri.u-bordeaux.fr/∼dutour/CS/.

[4] A. Denise, I. Dutour, and P. Zimmermann. CS: a package for counting and generating combinatorial structures. *mathPAD*, 8(1):22–29, 1998. http://www.mupad.de/mathpad.shtml.

[5] J. Duran and S. Ntafos. An evaluation of random testing. *IEEE transactions on Software Engineering*, SE-10:438–444, July 1984.

[6] *Website of the ECL^{i}PS^e Constraint Logic Programming System*. http://www.icparc.ic.ac.uk/eclipse/.

[7] P. Flajolet, P. Zimmermann, and B. Van Cutsem. A calculus for the random generation of labelled combinatorial structures. *Theoretical Computer Science*, 132:1–35, 1994.

[8] S.-D. Gouraud. Application des techniques de génération aléatoire au test de logiciel, january 2001. D.E.A. Programmation report, revised version.

[9] T. Hickey and J. Cohen. Uniform random generation of strings in a context-free language. *SIAM. J. Comput*, 12(4):645–655, 1983.

[10] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.

[11] B. Marre and A. Arnould. Test sequences generation from LUSTRE descriptions : GATEL. In *15h I.E.E.E. International Conference on Automated Software Engineering*, pages 229–237, 2000.

[12] J. Musa, A. Iannino, and K. Okumoto. Software reliability: Measurement, prediction, application McGraw-Hill, 1987.

[13] A. Pretschner. Classical search strategies for test case generation with constraint logic programming. *Proceedings of the Workshop on Formal Approaches to Testing of Software (FATES '01)*, pages 47–60, august 2001.

[14] The MuPAD Group (Benno Fuchssteiner et al.). *MuPAD User's Manual - MuPAD Version 1.2.2 Multi Processing Algebra Data Tool*. John Wiley and sons, 1996. http://www.mupad.de/.

[15] P. Thévenod-Fosse and H. Waeselynck. An investigation of software statistical testing. *The Journal of Software Testing, Verification and Reliability*, 1(2):5–26, july-september 1991.

[16] P. Thévenod-Fosse and H. Waeselynck. Statemate applied to statistical software testing. *ACM International Symposium on Software Testing and Analysis (ISSTA)*, pages 99–109, june 1993.

[17] P. Thévenod-Fosse, H. Waeselynck, and Y. Crouzet. An experimental study on software structural testing: deterministic versus random input generation. *21st IEEE Annual International Symposium on Fault-Tolerant Computing (FTCS'21)*, pages 410–417, 1991.

[18] L. White. Basic mathematical definitions and results in testing. In B. Chandrasekaran and S. Radicchi, editors, *Computer Program Testing*, ISBN 0-444-86282-7, pages 13–24. North-Holland, June 29-July 3 1981.