RAPPORT DE RECHERCHE

L R I

AuGuSTe : A TOOL FOR STATISTICAL TESTING EXPERIMENTAL RESULTS

GOURAUD S D

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud – LRI

03/2005

Rapport de Recherche N° 1400

CNRS – Université de Paris Sud
Centre d'Orsay
LABORATOIRE DE RECHERCHE EN INFORMATIQUE
Bâtiment 490
91405 ORSAY Cedex (France)

AuGuSTe: a Tool for Statistical Testing Experimental results

S.-D. Gouraud, gouraud@lri.fr
L.R.I., UMR 8623,
CNRS and Université Paris XI,
91400 Orsay, France.

Abstract

In a previous paper, we described a new generic method for statistical testing of software procedures, according to any given graphical description of the behavior of the system under test (control flow graph, statecharts, etc.). Its main originality is that it combines results and tools from combinatorics (random generation of combinatorial structures) with symbolic constraint solving, yielding a fully automatic test generation method. Instead of drawing input values as with classical testing methods, uniform random generation routines are used for drawing paths from the set of possible execution paths or traces of the system under test. Then a constraint solver is used for finding actual values for activating the generated paths.

In this paper, we present a first application of our method to structural statistical testing, first defined by Thevenod-Fosse and Waeselynck (LAAS), and the tool we have developed. We also present the experiments (more than 10000 on four functions from an industrial software) that we made in order to assess our approach and its stability. They show that our approach is comparable to the one of the LAAS, is stable and has the additional advantage of being fully automated. Moreover, these first experiments show that the method scales up well. The experiences gained from these experiments are also described and we present some perspectives.

1 Introduction

Generally, software testing consists in selecting test data, running the software and verifying that the outputs or the observed behaviour are in conformity with what was expected. The different approaches to select test data fall in three families: the structural (white-box) methods which select test data sets according to the code of the program, the functional (black-box) methods which select test data sets according to the specification of the system and the statistical methods which randomly select test data sets according to a distribution, usually over the input domains. Classically, this distribution is uniform [2, 11, 19] but it can also be derived from the operational profile of the environment of the future system [18].

When the statistical testing methods can be automated, they are used to draw large number of test data, allowing more intensive test campaigns than structural or functional testing methods. Unfortunately, some particular cases like exception cases are not or poorly covered.

To improve this technique, Thévenod-Fosse and Waeselynck from LAAS (Toulouse, France) defined a new kind of testing method which combines statistical methods with structural [22] or functional methods [23]. Their method provides a way to combine random testing and coverage requirements, in such a way that no element in a given set to be covered will be seldom or never exercised during testing. It 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 criterion, maximises 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 avoids a too weak probability for any 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 criterion 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.

Inspired by their work, we proposed a new approach [16] of structural statistical testing that we recently extended to any statistical method [10] according to any graphical description of the behaviour of the system under test (control flow graph, statecharts, etc.). Its main originality is that it combines results and tools from combinatorics (random generation of combinatorial structures) with symbolic constraint solving, yielding a fully automatic test generation

method. Instead of drawing *input values* as with classical testing methods, uniform random generation routines are used for drawing *paths* from the set of possible execution paths or traces of the system under test. Then a constraint solver is used for finding actual values for activating the generated paths.

In order to validate our approach, we developed the AuGuSTe tool and we performed many experiments with the following objectives:

1. evaluate the fault detection power of our approach. We compared for a set of programs the number of faults found by our approach and by the LAAS method.
2. evaluate the stability of the fault detection power. As with all random methods, we need to be sure that our results are not due to a lucky drawing but that they stay similar from one experience to another.
3. check if the method scales up well. Many methods are efficient on small programs but become impracticable on real size programs.

This paper presents the experiences we performed to validate our approach. In a first section, we recall briefly our approach, applied on structural statistical testing. The second section presents the AuGuSTe tool. Section 3 gives the context (programs, number of tests, etc.) of our comparison. Then in Sections 4 and 5, we present our experimental results. Finally, we conclude on these experiences and we give some perspectives.

2 Our structural statistical testing method

Classically, a program is associated with its control flow graph [1]. From the control flow graph G of the program under test, our approach consists in generating paths of G in order to cover a given criterion. Paths are generated by reusing tool for combinatorial structures: the MuPAD-Combinat package. The Mupad-Combinat[25] of MuPAD[20] includes the CS [3, 9] package which is devoted to counting and randomly generating combinatorial structures, based on the general notion of "decomposable structures" defined in [13].

Let C a structural coverage criterion, let E_C a set of elements of G which satisfy C and let N the desired number of tests. In the case of statistical testing, the satisfaction of a coverage criteria C by a testing method is characterised by the minimal probability $q_{C,N}$ of covering any element of E_C when drawing N tests. In [21], $q_{C,N}$ is called the test quality of the method with respect to C.

2.1 Paths generation

The first step consist in drawing a set of N paths of G such as the test quality is maximal.

If E_C denotes a finite set of paths of G, as for the criterion "all paths of length less or equal than n", the test quality is maximal if the paths of E_C are uniformly randomly generated.

Otherwise, if E_C is not described as a set of paths but as a set of elements of G (nodes, edges, circuits, etc.), we must maximise the minimal probability of covering any element of E_C when drawing N tests something that is not guaranteed by a uniform random generation of paths. The generation of paths becomes more complicated and is made in two steps:

1. Randomly choose N elements e_1, ..., e_N of E_C according to a distribution on the elements which maximise the test quality. In [15], we propose three different distributions on the elements: a uniform distribution, a distribution based on the notion of dominator in a graph [16] and a distribution based on the resolution of a linear programming system which allows to maximise the minimal probability of reach an element [10]. Since the uniform distribution was not satisfactory in practice, it was discarded for the experiments.
2. For each element e_i drawn in 1, uniformly randomly generate a path among all paths of G which pass through e_i.

In [24], the structural statistical method is different from our approach, since it is based on the explicit construction of a distribution on the input domain, either analytically, or empirically. In our case, we draw paths before using constraint-solving tools to produce the inputs. Of course, this induces a distribution on the input domain. As this distribution is highly dependent on the implementation of the constraint solver, it remains implicit. Therefore, the only way to compare the two methods is by experimentally comparing their fault detection power.

2.2 From paths to input data

When a set of N paths is obtained, the next step consist in deducing input data which allow to execute exactly these paths when it is possible (some paths cannot correspond to any run). For each path, the predicate characterising the input data which cause the execution of this path, is built and we try to solve it with constraint solving techniques. Unfortunately, in the general case, the predicate resolution is an undecidable problem.

3 The AuGuSTe tool

AuGuSTe is the tool we have developed for experimenting the method described in Section 2. Its modular architecture allows for an easy switch of the programming language

of the programs to test, the constraint solver and the distribution on the elements to be used. Developed in Objective Caml, it uses several extern packages in Java, Prolog or C language.

AuGuSTe takes four input data: a program P to test, a coverage criterion C, a number of tests N and a maximal length[1] n of paths. Currently, the program P is written in a simple imperative language inspired from C and Pascal. The basic constructions are sequential composition, *If...Then...Else* construction (*Else* is optional), *While* loop and *For* loop. The data types we consider are booleans, integers, arrays of booleans and arrays of integers. The criterion C is chosen among "all paths of length $\leq n$", "all branches" and "all statements".

The outputs are the N paths with their corresponding test data, a file containing the combinatorial structures and, possibly, another file containing the distribution on the nodes (resp. edges).

AuGuSTe proceeds in three main steps: the analysis, the paths generation and the resolution.

3.1 The analysis step

This first step consists in building the control flow graph G of the program P.

If C is "all paths of length $\leq n$" then a representation of all paths of G is built. This set can be easily translated into a combinatorial structure. The restriction on the length of paths is done with a parameter of the generation function of MuPAD-Combinat. One of our Objective Caml package is devoted to translating a graph to a corresponding combinatorial structure.

If C is "all statements" (resp. "all branches") then a distribution on the nodes (resp. edges) is built and for each node (resp. edge), the representation of all paths of G passing through this node (resp. edge) is built. The distribution on the nodes (resp. edges) can be calculated by the method based on dominators or by the method based on the resolution of a linear programming system evoked in Section 2.

For the distribution based on the resolution of a linear programming system, additional constructions are necessary: we have to know for each element e_i how many paths go through e_i and for each couple of elements (e_i, e_j) how many paths go through both e_i and e_j. Finally, the linear programming system is solved by an optimisation function using a simplex algorithm of MuPAD.

3.2 The paths generation step

This second step consists in randomly generating paths in one or two steps. If the criterion is "all paths of length $\leq n$" then N paths of length $\leq n$ are uniformly randomly generated. If the criterion is "all statements" (resp. "all branches"), N nodes (resp. edges) are randomly generated according to the calculated distribution. Then for each node (resp. edge) drawn, a path among all the paths of length $\leq n$ passing through this node (resp. edge) is randomly generated.

The random generation of the elements is performed using the random function of Objective Caml. First, a partition of the interval $[0; 1[$ is done according to the set of the elements probabilities. Then N float numbers are drawn and the N elements are deduced from them. For example, suppose we have two elements e_1 and e_2 with probabilities $p(e_1) = \frac{1}{4}$ and $p(e_2) = \frac{3}{4}$, if the drawn float number is between $[0; 0.25[$ then it is e_1 that is chosen and if the drawn float number is between $[0.25; 1[$ then it is e_2 that is chosen.

3.3 The resolution step

This third step consists in first building the predicates corresponding to each path then trying to resolve them.

Each path predicate, which is a conjunction of boolean expressions, is translated into logical constraints and a constraint solving package is used to compute a solution of the resulting constraint system. This package is borrowed from the GATeL tool [17] which automates test sequences generation from LUSTRE descriptions.

The constraint solver is built over finite domain libraries and coroutining mechanisms provided by the ECL^{i}PS^e environment [12]. Boolean variables take values inside the boolean domain, boolean operations are handled by symbolic simplifications and unification, integer variables take values inside unions of integer intervals (bounded by "minInt" and "maxInt"), and arithmetic operations and comparisons are handled by reductions of interval domains and unification.

When the resolution of all the predicates is completed, the predicates are sorted in three categories according to their results:

1. The predicate is found to be satisfiable: it is the easiest case. Each solution is a test data for executing the associated path.
2. The predicate is found to be unsatisfiable (the resolution fails): the associated path is proved unfeasible.
3. The other predicates: we do not know if they have or not a solution. To provide solutions in a bounded and reasonable time, the resolution is limited by a number of backtracks and by the time per attempt. When these bounds have been reached without finding a solution, the predicate resolution is aborted. In our experiments, this represents less than 1% of a set of 10^{16} paths.

The AuGuSTe tool draws a new path each time a predicate falls in one of the last two categories, until N test data are

[1] The choice of a right n will be evoked in Sections 5 and 6.

generated. For this step, two strategies are possible. For a given path, the first one consists in going back to the step of *drawing elements* while the other one consists in recovering the element e_i associated with the path. Then, in both cases, we *draw a path* among all paths going through the element. Both solutions are implemented in AuGuSTe but the experiments were done with the first one.

Contrarily to classical constraint solvers, our solver uses randomised resolution i.e. variables are randomly instantiated [16]. This kind of resolution is a feature with two advantages. First, when a same path is generated several times, we can have different input data to execute this path, something really important in software testing. Second, whenever the resolution of a predicate aborts, if this predicate has indeed a solution, there are more chance that this solution will be obtained in a following attempt if the same path is generated again.

4 Our experiments

In this section, we recall the principles of mutation testing. Then, we present the context (programs, mutants, coverage criteria, number of tests) of our experiments.

4.1 Mutation testing

Classically, mutation testing is used as a selection method [8], but it can be also used to evaluate the efficiency of dynamic testing generation methods [14]. It is a method of fault injection which consists in creating mutants of the program under test, i.e. clones of the program in which only one elementary error is introduced [7]. Given a test data set, each mutant is executed on the test data set and there are two possible cases:

1. On at least one test data the program and the mutant have different outputs. The mutant is said to be killed by the test data set.
2. For each test data, the program and the mutant have exactly the same outputs. The test data set is said to fail to kill the mutant.

In the last case, we need to determine if the mutant is an equivalent mutant or not. An equivalent mutant is a mutant for which no test data can distinguish the mutant from the original program. The equivalence of two programs being an undecidable problem, the determination of the equivalence is made by hand.

In some cases, the equivalence of some mutants can depend on the environment (operating system, compiler or versions of a compiler). Their equivalence can change between two different execution environments or even between two executions in a same environment. A particular case is when the program uses non-initialised variables or produces bad memory references, whose handling differ widely between environments. This will be the case in our experiments.

The second weakness of mutation testing is the high number of mutants. Indeed, we need to build a mutant for each operation, each variable, each type, etc. Rapidly, we obtain an impracticable number of mutants for a program. Mutants must be chosen in an intelligent way. Some tools like Mothra [6] or SESAME [4] automatically generate a mutants set of a program.

A test data set, and by extension the method which created this set, is evaluated by measuring the proportion of non equivalent mutants which are killed. This proportion is called the mutation score [26, 5]. It is a number between 0 and 1: a high mutation score indicates that the test data set is very efficient.

4.2 Programs and mutants

The experiments were performed to validate our approach: evaluate its fault detection power, its stability and check if it scales up well. As we wrote in Section 2, the only way to compare our approach and the method of the LAAS is by experiments on their fault detection power: we have to perform the same experiments as in [26]. We exploited the mutants and programs, Thévenod-Fosse, Waeselynck and Crouzet used to evaluate structural statistical testing method [24]. Thanks to them, it was possible to reuse the same sets of mutants and to replay almost the same set of experiments. For more details, we refer to [26, 24].

The experiments were performed on four C functions (FCT1, FCT2, FCT3 and FCT4) from an industrial software. Table 1 presents the profile of each function i.e. its number of code lines, and the number of paths, of blocs, of edges and of choice points (*While*, *IfThen*, *IfThenElse*) in its control flow graph.

	♯lines	♯paths	♯blocs	♯edges	♯choice pts
FCT1	30	17	14	24	5
FCT2	43	9	12	20	4
FCT3	135	33	19	41	12
FCT4	77	∞	19	41	10

Table 1 : *The four functions under test*

For FCT1, FCT2 et FCT3, the strongest structural criterion, i.e. “all paths”, was performed because all these functions have a finite number of paths. Since FCT4 contains a loop, the weaker criterion “all branches” was chosen.

The number of tests needed for each function was calculated in order to obtain a test quality of 0.9999. Only one set of tests was performed on the simple functions FCT1 and FCT2. More sets were done for FCT3 and FCT4 because

FCT3 is strongly dependent of the environment (some variables in the program are not initialised) and FCT4 contains a loop. This challenges the stability of the methods.

Table 2 summarises the number of runs performed for each function.

	criterion	♯sequence	♯tests N per sequence
FCT1	all paths	1	170
FCT2	all paths	1	80
FCT3	all paths	5	405
FCT4	all branches	5	850

Table 2 : *Number of tests*

The experiments were performed on 2914 mutants obtained with SESAME. For each function, the number of mutants is different according to the code complexity and the length: there are 279 mutants of FCT1, 563 of FCT2, 1467 of FCT3 and 605 of FCT4. There are three kinds of mutation: constant mutation, operator mutation and symbol mutation.

5 Experiments on FCT1, FCT2 and FCT3

Our tool needs to determine an upper bound of the paths length. As FCT1, FCT2 and FCT3 have a finite paths set, this bound correspond to the longest path length of the control flow graph. Hence, for FCT1 and FCT3 we consider "all paths of length ≤ 16" and for FCT2 we consider "all paths of length ≤ 14".

Table 3 gives the results for these three functions by the reference method [26] and by our tool.

	mutation score	
	[26]	AuGuSTe
FCT1	min=1 ave=1 max=1	min=1 ave=1 max=1
FCT2	min=1 ave=1 max=1	min=1 ave=1 max=1
FCT3	min=1 ave=1 max=1	min=0.9951 ave=0.9989 max=1

Table 3 : *Experimental results for FCT1, FCT2 and FCT3*

For FCT1 and FCT2, which are simple functions, we obtain exactly the same perfect mutation score as the reference method: all non equivalent mutants are killed.

For FCT3, the presence of non initialised variables are accountable for the great environmental dependence of the runs and particularly of the order in which the tests are executed. Whatever the testing method, to find all errors in a program, which is strongly "environment dependent", is not an easy task because the exact behaviour of the program is not defined. Indeed, its behaviour is not predictable for each test data set, and worse, we cannot guarantee that this behaviour stays stable if the program is executed several times on the same test data set. Note that this kind of problem can be detected before any dynamic test by a static data flow analysis.

That is why all test data sets have neither the perfect mutation score nor the same mutation score. As we do not have the same environment as the LAAS, it is impossible to know if they would obtain again perfect mutation scores in our environment.

Note that if all variables are initialised, then all our test data sets have a perfect mutation score.

6 The particular case of FCT4

FCT4 is the most interesting function. Indeed, it is the only one which has a loop and therefore an infinity of paths. It brings some problems that lead to several experiments and improvements of AuGuSTe.

First, we present the properties of this function, then we present the different experiments and the results we obtained.

6.1 Properties of FCT4

The code of FCT4 is divided in two parts:

- A conditional instruction which sets the variable `NB_VOIES` at the value 18 or 19;
- A loop with `NUM_VOIES<NB_VOIES` as loop condition. The variable `NUM_VOIES` is initialised to 0 and incremented by 1 at each iteration.

A path which does not exactly perform 18 or 19 iterations is unfeasible.

Moreover, the loop has another particularity: it contains a *IfThenElse* instruction with the condition `SE_VOIE_EN_TEST=TRUE`, where `SE_VOIE_EN_TEST` is an input variable that never changes during an execution. This conditional instruction is a source of many unfeasible paths too.

Note that all these characteristics can be found again with a suitable static code analysis.

As previously, we must determine an upper bound for the length of paths that allows to randomly generate all elementary paths without covering unnecessarily too many paths.

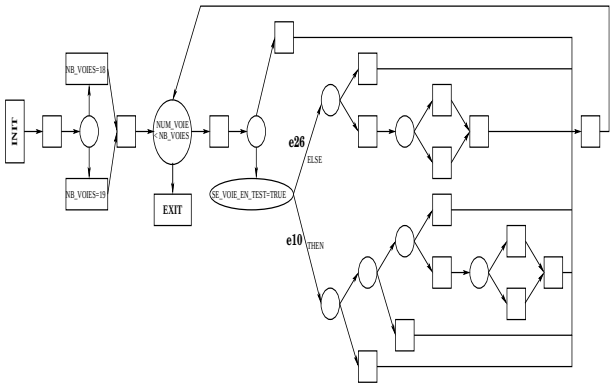

Figure 1 : *Control flow graph of FCT4*

6.2 Selection of n

First, we have to select the upper bound for the length of paths of the control flow graph of FCT4 (see Figure 1). A constraint on this length is that we have to be able to draw all paths performing 18 or 19 iterations. To calculate the upper bound, we need three values:

- $l_1 = 5$: length of the longest elementary[2] path from `INIT` (the root of the control flow graph) to the loop;
- $l_2 = 12$: length of the longest elementary paths inside the loop;
- $l_3 = 1$: length of the longest elementary path from the loop to `EXIT`.

The upper bound for the length of paths is $5 + 12 \times 19 + 1$ i.e. $n = 234$. Our aim is therefore to find five sets of 850 test data that cover "all branches" in randomly generating paths of length less or equal to 234.

6.3 Experiments and results

When building the combinatorial structure associated with the control flow graph, we can either ignore the characteristics of the program, or try to reflect them in the combinatorial structure. First, we chose to ignore them but as we will see that was not a good choice in this example. Indeed, that represents more than 10^{30} paths of length ≤ 234 with more than 99,98% of unfeasible paths. As we soon learned, it was unrealistic.

6.3.1 Specifying the number of loop iterations to the combinatorial structures

To reduce the number of paths, we added the knowledge about the number of loop iterations to the combinatorial structure. Then only the paths of length ≤ 234 which pass 18 or 19 times through the loop could be randomly generated. That represents 10^{20} paths.

For the two distributions evoked in Section 2.1, the minimal probability to reach an edge is $\frac{1}{2}$. Unfortunately, all test data sets which were generated did not cover the criterion "all branches".

The number of unfeasible paths and their distribution are the cause. Over the 4 232 437 paths drawn during an experience, 4 231 587 were unfeasible: that represents still around 99,98% unfeasible paths. Moreover, the probability to draw a feasible path reaching edge $e10$ is much greater than the probability to draw a feasible path reaching edge $e26$. This high proportion of unfeasible paths is a major problem for this case. One of the sources of these unfeasible paths is identifiable by static analysis: it is the condition instruction with the predicate constant value.

6.3.2 Optimisation of the combinatorial structure

To reduce still further the number of unfeasible paths, we have virtually transformed the control flow graph into another one when translating FCT4 into a combinatorial structure. This transformation is similar to moving constant computation out of loop in compiler optimisation[1].

Figure 2 shows the associated graphical description. Note that we changed the translation into a combinatorial structure, not the program itself!

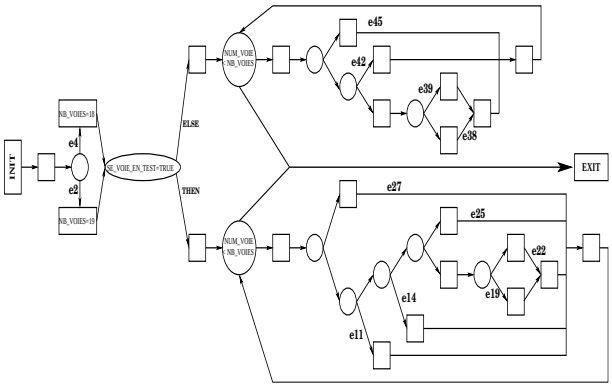

Figure 2 : *The new virtual control flow graph of FCT4*

The experiments prove this modification to be very interesting: both the number of paths and the number of unfeasible paths decrease. There remains 10^{16} paths of length ≤ 234 with now only 50% unfeasible paths.

For AuGuSTe with the distribution based on dominators, AuGuSTe(1), the minimal probability to reach an edge is 0.3324 and the distribution on the edges was:

$$\begin{cases} \pi_{e2} = \pi_{e4} = \pi_{e11} = \pi_{e14} = \pi_{e17} = \pi_{e22} = 0.8333 \\ \pi_{e25} = \pi_{e27} = \pi_{e38} = \pi_{e39} = \pi_{e42} = \pi_{e45} = 0.8333 \\ \text{and } \pi_x = 0 \text{ for all other edges} \end{cases}$$

[2] An elementary path is a path which pass at most one time by an edge.

As the criterion "all branches" was not always covered, we analysed the generated paths. We discovered that 4 test data sets do not contain any feasible path passing through the *Else* branch of the instruction *IfThenElse* with the predicate `SE_VOIE_EN_TEST=TRUE`. The fifth test data set has only one such path.

In fact, the numbers of paths passing through the *Else* branch and the *Then* branch are still too much dissymmetrical:

- 687×10^9 paths pass through the branch where the predicate `SE_VOIE_EN_TEST=TRUE` is false
- 10^{15} paths pass through the branch where the predicate `SE_VOIE_EN_TEST=TRUE` is true

The distribution on the edges does not restore the balance.

For AuGuSTe with the distribution based on the linear programming system, AuGuSTe(2), the minimal probability to reach a edge is 0.4923 and the distribution on the edges was:

$$\begin{cases} \pi_{e11} = \pi_{e14} = \pi_{e17} = \pi_{e22} = \pi_{e25} = \pi_{e27} = 0.0844 \\ \pi_{e38} = \pi_{e39} = \pi_{e42} = \pi_{e45} = 0.1235 \\ \text{and } \pi_x = 0 \text{ for all edges} \end{cases}$$

The results are better than with AuGuSTe(1) and this time, the criterion "all branches" is always covered. By giving higher probabilities to the edges of the small loop (i.e. where there are less paths), the balance is restored.

	mutation score		
	[26]	AuGuSTe (1)	AuGuSTe (2)
FCT4	min=0.9898 ave=0.9901 max=0.9915	min=0.9726 ave=0.9773 max=0.9854	min=0.9854 ave=0.9854 max=0.9854

Table 4 : *Experimental results for FCT4*

Table 4 presents the results obtained with AuGuSTe and with the reference method of the LAAS. We can see that all results are comparable to the ones of the LAAS up to the slight difference in mutants equivalence. But AuGuSTe(1) and AuGuSTe(2) are automatically obtained.

7 Conclusion and Perspectives

This paper presents the AuGuSTe tool which implements an application to structural statistical testing of the generic statistical testing method proposed in [10] and our first experiments results.

The originality of this tool is based on random generation of combinatorial structure techniques to draw paths, on randomised constraint solver to solve the predicate, but also on its modularity. Indeed, the AuGuSTe tool can be adapted to any constraint solver (compatible with the semantic of the graph), to any paths generator and to any static analysis package which translates a program into a graph. Moreover, to our knowledge, it is the first tool for structural statistical testing.

The experiments show that our approach is comparable to the one of LAAS, is stable and has the additional advantage of being completely automated. The functions used for these experiments are characteristic of realistic unit functions: if we unfold the code of FCT4 according the loop iteration property, we obtain a function of 290 code lines. Contrarily to classical approaches, based on a distribution on the input domain, which have no problem with unfeasible paths but have problems of coverage, the hard point of our approach is the unfeasible paths. The ratio of unfeasible paths probably matters more than the length of the code or other structural complexity measures. FCT4 with its 99,98% of unfeasible paths was rather challenging in that respect. This lets us think that the method scales up well.

More generally, this approach could provide a basis for a new class of tools in the domain of software testing, combining random generation of combinatorial structures, linear programming techniques, and constraint solvers.

Different perspectives are open: some are directly linked to the approach but we do not discuss about them here (see [10]). The others are optimisation and extension of the tool.

The analysis step (construction of the combinatorial structure and counting the number of paths) is very costly, particularly for calculating the distribution based on a linear programming system. Currently, it takes 15 minutes for FCT4. It can be improved by avoiding unnecessary constructions and optimising combinatorial structures. We are working in this direction.

The test generation time (drawing paths and solving predicates) takes two hours for generating 850 feasible paths of FCT4 among 10^{16} paths with 50% unfeasible paths. Solving predicates is more expensive than drawing paths. But the presence of unfeasible paths is still expensive because we loose time to try solving unfeasible paths and to repeat paths generation step and resolution step.

Unfeasible paths have also effects on the distribution[10] as we saw in Section 6. To limit their number, a static analysis could be done when it is necessary (many paths, presence of loops) or/and it could take into account some annotations of the programmer in order to obtain an elaborated combinatorial structure closed to the semantic or operational profile of the program.

We might also use grammatical inference to learn from a set of generated paths, that we know to be feasible, unfeasible or indeterminate, some properties like "all paths passing three times in the loop are unfeasible" then use that information to improve again the combinatorial structure.

Some works are also going towards an human interaction

where the user will directly select a set of nodes or edges and describe his own criterion, or adjust the translation into a combinatorial structure as we have shown with FCT4 by unfolding some part.

Acknowledgments

We wish to acknowledge : the testing group in LAAS for providing us the library of mutants, Sylvie Corteel and Nicolas Thiery for the combinatorial structures library, Bruno Marre for his help in extending and using the constraint solver and Frédéric Voisin for all his highly useful comments on this paper. This work was partially funded by the European community (IST Project 1999-11585: DSoS).

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1988. ISBN 0-201-10088-6.

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

[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] Y. Crouzet, P. Th´evenod-Fosse, and H. Waeselynck. Validation du test du logiciel par injection de fautes : l'outil SESAME. In *11ème Colloque National de Fiabilité & Maintenabilité*, pages 551–559, 1998.

[5] R. DeMillo. Mutation analysis as a tool for software quality assurance. In *Proceedings COMPSAC'80*, pages 390–393, 1980.

[6] R. DeMillo, D. Guindi, K. King, W. McCracken, and A. J. Offutt. An extended overview of the Mothra software testing environment. In *Proceedings of the 2nd Workshop on Software Testing, Verifi cation, and Analysis*, 1988.

[7] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: help for the praticing programmer. *IEEE Computer Magazine*, 11(4):34–41, avril 1978.

[8] R. DeMillo and A. Offut. Constraint-based automatic test data generation. *IEEE Transaction on Software Engineering*, 17(9):900–910, septembre 1991.

[9] 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.

[10] A. Denise, M.-C. Gaudel, and S.-D. Gouraud. A generic method for statistical testing. In *Proceedings of the 15th. IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2004 (to appear).

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

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

[13] 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.

[14] M.-C. Gaudel, B. Marre, F. Schlienger, and G. Bernot. *Précis de génie logiciel*. Masson, 1996. ISBN 2-225-85189-1.

[15] S.-D. Gouraud. *Utilisation des Structures Combinatoires pour le Test Statistique*. PhD thesis, Universit´e Paris XI, Orsay, june 2004.

[16] S.-D. Gouraud, A. Denise, M.-C. Gaudel, and B. Marre. A new way of automating statistical testing methods. In *IEEE International Conference on Automated Software Engineering*, pages 5–12, 2001.

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

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

[19] S. C. Ntafos. On comparisons of random, partition, and proportional partition testing. *IEEE Transactions on Software Engineering*, 27(10):949–960, october 2001.

[20] 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/.

[21] P. Th´evenod-Fosse. Software validation by means of statistical testing: Retrospect and future direction. In *International Working Conference on Dependable Computing for Critical Applications*, pages 15–22, 1989.

[22] P. Th´evenod-Fosse and H. Waeselynck. An investigation of software statistical testing. *The Journal of Software Testing, Verifi cation and Reliability*, 1(2):5–26, july-september 1991.

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

[24] P. Th´evenod-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.

[25] N. M. Thi´ery. Mupad-combinat – algebraic combinatorics package for mupad. http://mupad-combinat.sourceforge.net/.

[26] H. Waeselynck. *Vérifi cation De Logiciels Critiques Par Le Test Statistique*. PhD thesis, Institut National Polytechnique, Toulouse, 1993.