HAL
open science

Distinguish Dynamic Basic Blocks by Structural Statistical Testing

Matthieu Petit, Arnaud Gotlieb

To cite this version:

Matthieu Petit, Arnaud Gotlieb. Distinguish Dynamic Basic Blocks by Structural Statistical Testing. 12th European Workshop on Dependable Computing, EWDC 2009, May 2009, Toulouse, France. 8 p. hal-00381557

HAL Id: hal-00381557

https://hal.science/hal-00381557v1

Submitted on 12 May 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distinguish Dynamic Basic Blocks by Structural Statistical Testing

Matthieu Petit*
Departement of Communication, Business and Information Technologies
Roskilde University
P.O Box 260, 4000 Roskilde Denmark
petit@ruc.dk

Arnaud Gotlieb
INRIA Rennes - Bretagne Atlantique
Campus Beaulieu
35042 Rennes cedex, FRANCE
Arnaud.Gotlieb@irisa.fr

Abstract

Statistical testing aims at generating random test data that respect selected probabilistic properties. A distribution probability is associated with the program input space in order to achieve statistical test purpose: to test the most frequent usage of software or to maximize the probability of satisfying a structural coverage criterion for instance.

In this paper, we propose a new statistical testing method that generates sequences of random test data that respect the following probabilistic properties: 1) each sequence guarantees the uniform selection of feasible paths only and 2) the uniform selection of test data over the subdomain associated with these paths. Baudry et al. present a testing-for-diagnosis method where the essential notion of Dynamic Basic Block was identified to be strongly correlated to the effectiveness of fault-localization technique. We show that generating a sequence of random test data respecting these properties allows to well-distinguished Dynamic Basic Blocks. Thanks to Constraint programming techniques, we propose an efficient algorithm that uniformly selects feasible paths only by drastically decreasing the number of rejects (test data that activate another control flow path) during the test data selection. We implemented this algorithm in a statistical test data generator for Java programs. A first experimental validation is presented.

1 Introduction

Software testing is the first validation technique used to produce high-quality software, but it is also a primary cost driver on most projects. Software testing aims at detecting defects within programs, while debugging aims at locating faults and correcting them. The times when these two complementary activities were separated is hopefully behind us. Nowadays, modern test data generation techniques cannot ignore how the test data will be used in order to locate the defects. Since several years, many automated fault-localization techniques have been proposed to use information about the test data executions through control-flow monitoring introduced [12]. These methods, globally called coverage-based fault localization, use statement coverage and test results to locate the statements that are the most correlated to the defects. Recently, studies have been conducted to identify how test data generation techniques impact the effectiveness of coverage-based fault localization techniques [2, 1, 21]. In [2], Baudry et al. present a testing-for-diagnosis method where the essential notion of *Dynamic Basic Block* was identified to be strongly correlated to the effectiveness of any fault-localization technique. Roughly speaking, a *Dynamic Basic Block (DBB)* is the subset of statements in a program that are equally covered by the test data of a given test suite. Surprisingly, the experimental results of [2] show a strong correlation between the effectiveness of fault-localization technique and the size of the DBB that includes the faulty statement. They conclude that maximizing the number of DBBs increases the effectiveness of coverage-based fault-localization techniques.

As proposed by Thevenod-Fosse and Waeselynck [18], test data generation methods include deterministic and statistical testing. While deterministic testing aims at selecting a single test data in front of a given test objective, statistical testing generates test data at random. Statistical testing covers three distinct ways of generating test data:

- (*Random Testing* [7]) Test data are generated according an uniform distribution probability, meaning that each point of the input space of a program has the same probability to be selected ;
- (*Operational Testing* [14]) Test data are generated ac-

*This work is supported by the project GENETTA granted by Brittany region and the project "Logic-statistic modeling and analysis of biological sequence data" funded by the NABIIT program under the Danish Strategic Research Council.

cording to an operational usage profile of the program. Usually, this profile is specified by the tester and does not depend on the program itself ;

- (*Functional and Structural statistical testing* [18]) Test data are generated according to a distribution probability that is computed from a model (functional) or the program itself (structural). In the latter case, the goal is to maximize the coverage of a selected coverage criterion by maximizing the probability to exercise the least probable element of the criterion.

In this paper, we propose a new statistical test data generation technique that improves coverage-based fault-localization. We built on our previous works by unifying two techniques : uniform selection of feasible paths [17] and path-oriented random testing [10]. The technique we propose in this paper aims at generating random test data that respect two specific probabilistic properties :

- The generated test suite will activate uniformly all the feasible paths of the program ;
- The generated test suite will ensure that the subdomain associated with each feasible path will be uniformly activated.

Generating a test suite that follows these two properties is challenging as establishing path feasibility is undecidable in the general case and generating a test suite that uniformly covers a subdomain defined by constraints usually requires many rejections. In our work, we benefit from well-recognized Constraint Programming techniques to determine, in most case, path feasibility and obtain a tight over-approximation of the path constraint solutions. We model uniform selection of feasible paths as an optimization problem. To cover a path, constraint propagation and refutation allow us to drastically decrease the number of rejects (test data following another path) while maintaining uniformity. Thanks to our implementation, our experimental results show that generating a sequence of random test data respecting these properties permits unambiguously to increase the number of well-distinguished DBBs, improving so coverage-based fault localization.

Outline of the paper. The paper is organized as follows : section 2 motivates our work on program **trityp**. Section 3 presents our statistical testing technique. Our implementation prototype GENETTA is described section 4 while section 5 presents our experimental validation. Finally, section 6 concludes this work.

2 Motivating Example

We illustrate our statistical test data generation method for fault-localization on an example. The program **trityp**, initially proposed by Myers [15] and fully studied by DeMillo and Offut [5], takes three non-negative integers as arguments that represent the relative lengths of the sides of a triangle and classifies the triangle as scalene (output equals to 1), isosceles (output equals to 2), equilateral (output equals to 3) or illegal (output equals to 4). The source code of a faulty version of the program is given by the **Figure** 1.

Although it implements a simple specification, this program is difficult to handle for test data generators as it contains several nested conditionals structures and a lot of infeasible paths (47 over a total of 57 in our version).

Random testing alone has some difficulties to achieve good coverage of the source code. Indeed, simple events such as `i=j` have low probability to happen ($\frac{1}{2^{32}}$ if i and j are 32-bits integers). Similarly, *statistical structural testing*, as implemented in [6], fails to achieve good coverage in a reasonable amount of time, as non-feasible paths are not discarded from the set of possible executions. In this example, 75% of selected paths are rejected after being generated.

In this paper, we propose a dynamic algorithm that generates sequences of random test data respecting both following probabilistic properties:

- each generated test suite guarantees the uniform selection of the 10 feasible paths of **trityp** ;
- each generated test suite for a given feasible path guarantees the uniform selection of test data over the associated sub-domain.

Figure1 shows the coverage matrix of two test suites: on the left side, test data are generated by Random Testing (RT), while on the right side, test data are generated according to our method. Both data have been generated automatically using a RT implementation and our implementation called GENETTA. In the conditional statement line 20, correct condition $j + k < i$ is replaced by faulty condition $j + k > i$. For sake of clarity, the input domain of each integer variable has been restricted to $0..1000$.

By definition extracted from [2], a dynamic basic block is a set of statements of a program that is covered by the same test data from a test suite. Two statements s and s' belong to a DBB if they have identical lines in coverage matrix.

From the RT coverage matrix, three *dynamic basic blocks (DBBs)* are extracted:

$$\{01, 03, 04, 06, 08, 10, 11, 23\}, \{12\}, \{13\}$$

while seven DBBs are extracted from GENETTA coverage matrix:

$$\{01, 03, 04, 06, 08, 10, 14, 16, 18, 23\}, \{05\}, \{07, 19\}, \{09\}, \{20\}, \{21\}, \{22\}.$$

	Random Testing Test Suite					Genetta Test Suite				
public int trityp(int i,int j,int k) **int t;**	216 635 762	319 239 330	848 686 271	830 265 979	274 684 31	344 344 838	50 50 932	817 660 817	240 68 68	914 666 666
01.**if**((i==0)||(j==0)||(k==0))	•	•	•	•	•	•	•	•	•	•
02. t=4;										
else{ 03. t=0;	•	•	•	•	•	•	•	•	•	•
04. **if**(i==j)	•	•	•	•	•	•	•	•	•	•
05 t=t+1;						•	•			
06 **if**(i==k)	•	•	•	•	•	•	•	•	•	•
07 t=t+2;								•		
08. **if**(j==k)	•	•	•	•	•	•	•	•	•	•
09 t=t+3;									•	•
10. **if**(t==0)	•	•	•	•	•	•	•	•	•	•
11. **if**((i+j<=k)||(j+k<=i)||(i+k<=j))	•	•	•	•	•					
12. t=4;					•					
else 13. t=1;	•	•	•	•						
else										
14. **if**(trityp>3)						•	•	•	•	•
15. t=3;										
else 16 **if**((t==1)&&(i+j>k))						•	•	•	•	•
17 t=2;										
else 18 **if**((t==2)&&(i+k>j))						•	•	•	•	•
19 t=2;								•		
else 20. **if**((t==3)&&(j+k<i))//< instead of >						•	•		•	•
21 t=2;									•	
else 22; t=4; }						•	•			•
23. return(t);	•	•	•	•	•	•	•	•	•	•
Test Verdict	**P**	**P**	**P**	**P**	**P**	**P**	**P**	**P**	**F**	**F**

Figure 1. Faulty version of trityp and coverage matrix

As a result, the DBB associated with the faulty statement (line 20) has size one for GENETTA while it is not activated with RT.

3 Constraint-based statistical test data generation

Constraint-based testing aims at modelling an automatic test data generation problem as a constraint problem [4, 9]. Relations between program variables and testing purposes are translated into constraints. In this section, we propose a new algorithm that guarantees the uniform selection of feasible paths only and the uniform activation of test data that follow each of these paths. Sec.3.1 describes the overall algorithm whereas Sec.3.2, 3.3 and 3.4 present step-by-step its distinct techniques. In this paper, we only consider programs composed of: *assignment statement*, *conditional statement*, *loop statement* and *compound statement*. Variables types are integer or array of integers. Note that the coverage of all execution paths is generally an intractable testing criterion, due to the presence of loops. As done in [20, 8], we limit the number of iterations in a *loop statement* to a fixed number k. Hence in the following, *loop statements* are considered as k imbricated *conditional statements*.

3.1 Algorithm of Statistical Test Data Generation

Statistical test data generation is described by **Algorithm** 1. It takes a program $Program$, Nb_DT the expected number of test data to generate and k the maximum number of iterations in loops as inputs, and issues a randomly generated test set.

The first step of the algorithm, line 2, corresponds to the translation of $Program$ and the testing purpose into a stochastic optimization problem $ctrs_proba$. The optimization function $Optim_Function$ is generated accordingly during this phase. This function implements the (currently unknown) number of feasible paths

of $Program$. Our approach allows a uniform selection of feasible paths without computing this number.

Algorithm 1: Statistical test data generation

Input : $Program$, Nb_DT, k
Output: DT_Seq

1 $DT_Seq \leftarrow \emptyset$;
2 Translate $Progran$ into $Ctrs_proba$ and generate $Optim_function$;
3 **while** `size(`DT_Seq`)`$< Nb_DT$ **do**
4 $\quad Path_Conditions \leftarrow$ Maximize $Optim_function$;
5 $\quad DT \leftarrow$ Select uniformly valuations that satisfy $Path_Conditions$;
6 $\quad$ Add DT in DT_Seq;
7 $\quad$ Update $Ctrs_proba$ in regards to the path activated by DT;
8 **end**
9 **return** DT_seq

The algorithm iterates until Nb_DT test data are generated. Each DT is generated in two steps (line 4 and 5). The search of the maximal value of $Optim_Function$ is performed as search over a tree, called the *probabilistic execution tree* [17]. The nodes of this tree are just the various basic blocks of $Program$ while its edges correspond to the possible transfer of control flow. In addition, the edges of this tree are labeled with the variables of the probability distribution.

Large parts of the tree can be removed during the stochastic optimization process when path conditions are detected as unsatisfiable. Searching an optimal value for $Optim_Function$ yields the selection of a feasible path, $Path_Conditions$.

The uniform selection of test data that activate a given path is performed by an efficient process we called *path-oriented random testing* in [10]. Constraint propagation leads to the computation of an over approximation of the subdomain associated with path conditions. Thanks to this result, we provide a divide–and–conquer algorithm that minimizes the number of rejects (test data that activate another path) while generating statistical test data. Note that checking whether a test datum follows the selected path, can be done on the constraint system itself just by verifying the consistency of its $Path_Conditions$. Finally, $Ctrs_proba$ is updated after each statistical test datum generation with the information on path feasibility.

3.2 Translation of the program into a constraint program

In our approach, the relations between program variables are modelled with constraints, as well as the statistical testing purpose. As said above, the generated constraint system is represented by a tree (*probabilistic execution tree*). In this section, we detail how each statement of the tested program can be translated within this tree.

3.2.1 Translating statements into constraints

Method Signature.
Signature of tested method does not appear in probabilistic execution tree. However, this signature is translated into a clause head[1]. Each variable of the signature is translated into a fresh variable and type information into a domain constraint. Body of the clause is composed of the probabilistic execution tree. For instance, input variable i of **trityp** is translated into I and type information by the constraint I in $-2^{31}..2^{31}-1$.
Sequence of assignment statements.
For each definition of a variable, a fresh variable is generated and a domain constraint to represent type information. Then, $var := Expr$ is translated into $X = E$ where E is the syntactic translation of $Expr$. For instance, **trityp** statement `t=t+1` is translated into $T_1 = T_0 + 1 \wedge T_1$ in $-2^{31}..2^{31}-1$. A sequence of assignment statements is translated into a conjunction of constraints.
Transfer of control flow.
Transfer of control flow associated with a conditional statement or a loop unfolding is translated into a $choose_decision(C, [W_1, W_2], Ctrs_1, Ctrs_2)$ where `C` is the syntactic translation of the decision, W_1 and W_2 represent the probability transitions to the two successor nodes in the tree and $Ctrs_1$ (resp. $Ctrs_2$) is the probabilistic execution tree associated with statements of the $true$ (resp. $false$) branch and remaining statements of the program. For example, first conditional of **trityp** is translated into:

$$choose_decision(I \leq 0 \vee J \leq 0 \vee K \leq 0, \\ [W_1, W_2], Ctrs_1, Ctrs_2)$$

where is a W_1 (resp. W_2) is a finite domain variable that represents the probability to add $(I \leq 0 \vee J \leq 0 \vee K \leq 0) \wedge Ctrs_1$ (resp. $(I > 0 \wedge J > 0 \wedge K > 0) \wedge Ctrs_2$) and $Ctrs_1$ (resp. $Ctrs_2$) is the probabilistic execution tree associated with $true$ branch (resp. $false$ branch) of a conditional statement.

3.2.2 Constraints on probability transitions

Suppose that the set of feasible paths of the probabilistic execution tree is known. A uniform selection of feasible paths can be easily performed. Indeed, the selection of feasible paths in the tree is uniform when probability transitions associated with each edge of this tree are proportional

[1] Constraint solver used is part of a logic programming language. That is the reason why some terminologies of this paradigm are used.

to the number of feasible paths that activate it. However in practice, the number of feasible paths which activates this edge is unknown. That is the reason why a couple of weight variables [W_1,W_2] is used to model probability transitions of the tree. Domain of weight variables represents information about the number of feasible paths that activate the edge. However, constraints on these weight variables can be generated from the tree structure.

Suppose that W_i is the weight variable associated with the edge which goes in a node and W_{o_1} and W_{o_2} the weight variables associated with the two branches which go out the node. The number of feasible paths that go in a node is equals to the number of feasible paths that go out. Then, a constraint $W_i = Wo_1 + Wo_2$ is generated for each probabilistic choice constraint. For instance, the number of feasible paths that activates the first conditional of **trityp** is equals to the sum of the number of feasible paths that active the $true$ and the $false$ branch.

Suppose that W_e represents a weight variable that leads to a leaf node. Domain of W_e is 0..1. Indeed, only one feasible path activates at most this edge. In the following of this paper, W_{exit} denotes the set of weight variables associated with leaf nodes of probabilistic execution tree.

3.3 A uniform selection of satisfiable path conditions only

Path selection is represented by generation of path conditions associated with a path of the probabilistic execution tree (from the root node to a leaf node). This path selection is performed when each probability transition that composes this path is valuated. Our approach is based on an iterative construction of the selector of path conditions. This iterative construction is represented by domain prunings of weight variables. Path selection is modelled as an optimization problem. Optimization function to maximize corresponds to the number of feasible paths, defined as follows:

$$\sum_{W_j \in W_{exit}} W_j.$$

A top-down labeling process on weight variables of W_{exit} is used to generate of path conditions. This labeling process leads to a valuation of weight variables. In that case, uniform path selection is performed on the set of feasible paths and the set of paths not yet detected as unfeasible. Decision procedure of the constraint solver allows us to detect as soon as possible that path conditions are not satisfiable. In that case, a backtrack mechanism is used to reload the search and find a new optimal valuation of the weight variables. Corresponding part of the probabilistic execution tree is also removed.

For instance, let us consider again the probabilistic choice constraint associated with the first conditional statement of **trityp**:

$$choose_decision(I \leq 0 \vee J \leq 0 \vee K \leq 0, \\ [W_1, W_2], Ctrs_1, Ctrs_2)$$

At the beginning, constraints generated from the tree structure constraint W_1 to 0..1 and W_2 to 0..56. These domains correspond to the potentially number of feasible paths that activate respectively the $true$ branch or the $false$ branch. First optimal solution for the $Optim_Function$ leads to a valuation to 1 for W_1 and 56 for W_2. This valuation can launch the generation of $I \leq 0 \vee J \leq 0 \vee K \leq 0$ as path conditions (path $01 - 02 - 23$). However, constraints

$$I > 0 \wedge J > 0 \wedge K > 0 \wedge I = J \wedge I = K \wedge J \neq K$$

associated with pre-conditions of $01 - 03 - 04 - 05 - 06 - 07 - 08$ can also be generated. As these constraints are not satisfiable, part of probabilistic execution tree associated with this edge is removed and the search is reloaded.

3.4 Uniform selection of test data that satisfy path conditions

In this section, we detail our algorithm to perform a uniform of test data that satisfy path conditions. In constraint programming terminology, this algorithm aims at selecting solutions of a constraint problem with the same probability. Determine this set of solutions is a NP-hard problem [11]. Rejection method [13] is a classical way to address this problem. Uniform selection is performed on an over-approximation of the solutions set. Valuation of variables is rejected when this valuation is not a solution of the problem. This valuation is kept when this valuation is a solution. However, rejection method can be inefficient when the sub-domain associated with input variables is too large w.r.t solutions of the path conditions. The algorithm presented in this section permits a trade off between the computation of solution sets and an inefficient rejection method. A divide and conquer is proposed to achieve this goal.

Firstly, we detail how to fairly divide the subdomain associated with path conditions resulting from constraint propagation (Sec. 3.4.1) and secondly, we explain how to exploit constraint refutation to prune this subdomain (Sec.3.4.2). Finally, we give our algorithm that exploits both these processes (Sec.3.4.3).

3.4.1 Dividing subdomain of path conditions

Applying constraint propagation on the path conditions results in an subdomain that is a correct approximation of the solution set of the path conditions. We propose a new way of refining this subdomain in a smaller one. Let div be a given parameter, called the *division parameter*, our method

is based on the division of each variable domain into div subdomains of equal area. When the size of a domain variable cannot be divided by div, then we enlarge its domain until its size can be divided by div. By iterating this process over all the n input variables, we get a fair partition of the (augmented) initial subdomain, in div^n subdomains.

3.4.2 Pruning subdomain of path conditions

Constraint refutation is the process of temporarily adding a constraint to a set of constraints and testing whether the resulting constraint system has no solution by using constraint propagation. If the resulting constraint system is unsatisfiable, the added constraint is shown to be contradictory with the rest of the constraints and then it is refuted. When constraint propagation does not yield to a contradiction, then nothing can be deduced as constraint propagation is incomplete in general. Based on constraint addition/removal and propagation, this process is very efficient and it can be exploited to reduce the over approximation of the set of solutions obtained by constraint propagation.

Constraint refutation is used to test efficiently domain intersection. Thus, we eliminate parts of the suddomain that are inconsistent with the path conditions. Constraint refutation has another advantage as it help detecting non-feasible paths. Recall that non-feasible paths correspond to unsatisfiable constraint systems. Hence, when all the subdomains of the partition are shown to be inconsistent, then it means that the corresponding path is unfeasible.

3.4.3 Algorithm

The algorithm takes as inputs a set of variables along with their variation domain, $Path_Conditions$ constraints set generated during path selection, div the division parameter, and N the length of the expected random sequence. The algorithm returns a list of N uniformly distributed random tuples that all satisfy the path conditions. The list is void when the corresponding path is detected as being non-feasible.

Firstly, the algorithm partitions the subdomain resulting from constraint propagation in div^n subdomains of equal area (`divide` function). Then, each subdomain D_i in the partition is checked for unsatisfiability. This results in a list of subdomains $D'_1, \ldots, D'_p$ where $p \leq div^n$. Secondly, uniform selection of test data is built from this list by picking up first a subdomain and then picking up a tuple inside this subdomain. If the selected tuple does not satisfy the path conditions then it is simply rejected. This process is repeated until a sequence of N test data is generated. This algorithm is semi-correct, meaning that when it terminates, it is guaranteed to provide the correct expected result, but it is not guaranteed to terminate. Indeed, in the second loop, N is decreased iff t satisfies $Path_Conditions$, which can happen only if $Path_Conditions$ is satisfiable. In other words, if $Path_Conditions$ is unsatisfiable and if this has not been detected by constraint propagation ($p \geq 1$), then the algorithm will not terminate. Note that similar problems arise with random testing or path testing as nothing prevents an unsatisfiable goal $Path_Conditions$ to be selected and, in this case, all the test cases will be rejected. In practice, a time out mechanism is necessary to enforce termination. This mechanism is not detailed here but it is mandatory on actual implementation. Note that any testing tools that execute programs should be equipped by such a time-out mechanism as nothing prevents the programs to activate endless paths.

Algorithm 2: Uniform selection of test data that satisfy path conditions

input : $(x_1, .., x_n)$, $Path_Conditions$, div, N
output: $t_1, .., t_N$ or $\emptyset$ (non-feasible path)

```
T := ∅;
(D_1, .., D_{k^n}) := divide({x_1, .., x_n}, k);
forall D_i ∈ (D_1, ..., D_{k^n}) do
    if D_i is inconsistent w.r.t. Path_Conditions then
        Remove D_i from (D_1, .., D_{k^n}) ;
    end
end
Let D'_1, .., D'_p be the remaining list of domains;
if p ≥ 1 then
    while N > 0 do
        Pick up uniformly D at random from
        D'_1, .., D'_p;
        Pick up uniformly t at random from D;
        if Path_Conditions is satisfied by t then
            add t to T;
            N := N − 1;
        end
    end
end
return T;
```

4 Implementation

Our prototype is implemented in SICStus Prolog (4500 LOC). The tool generates statistical test data for a restricted fragment of the JAVA language.

Given a JAVA class and a method of this class, our prototype generates a test suite at random that respects the probabilistic properties we defined above.

The implementation of the tool includes three main parts: source code parsing and analysis, constraint generation and constraint solving. The modules dedicated to source code analysis include a complete SUN's JAVA 1.4 parser that builds a symbol table and an abstract syntax tree. From this tree, the constraint generation modules de-

rive the *probabilistic execution tree* we introduced in Sec.3. Finally, the constraint solving modules implement probabilistic choice constraints which are the key-stone of the tool. These modules are built upon the clp(fd) [3] and the PCC(FD) [16] libraries of SICStus Prolog. The modules dedicated to constraint solving also implement the algorithms of Sec.3.1.

The user interacts with the tool through a batch mode that permits to ask various test data generation requests. The statistical test data generation can be parameterized through the predicate `genetta_launcher`. This predicate offers optional arguments for constraining the number of paths to be selected, the number of test data to be generated, k a parameter that bounds the loop unfolding, div the division parameter and $time - out$ a time-out value able to interrupt the generation. Note that this latter parameter is mandatory on every testing tool implementation as nothing guarantees the tested program to iterate indefinitely.

5 Experimental validation

In this section, we present our preliminary results on the impact of our statistical test data generator on the effectiveness of a standard coverage-based fault localization technique. Sec. 5.1 presents our experimental process whereas Sec.5.2 describes our results.

5.1 Experiments

The experimental process was divided into 4 steps:

1. Two sequences of test data were generated using GENETTA and a Random Testing implementation. Test data were generated for a correct version of the method under test;
2. Mutants were then generated for the method under test;
3. The generated test data were executed against all the mutants and the execution traces were collected within a coverage matrix;
4. Finally, the Dynamic Basic Blocks (DBBs) associated with both sequences of test data were computed;

This experimental process exploited two existing tools: MuJava [22] was used to generate a set of standard mutants and execution traces were collected thanks to the BUGDEL tool [19]. Finally, we implemented a predicate that computes DBBs associated with each test suite.

5.2 Experimental results

Trityp was used to serve as our candidate program. GENETTA and our RT implementation generated two sequences of 100 test data in order to defined the coverage matrix and 32 mutants of **trityp** were generated by MuJava. **Fig.**2 summarizes our results.

These results show unambiguously that our method improves the number of DBBs on every mutants. Interestingly, the maximum number of DBBs is reached on more than half of the mutants, indicating that GENETTA not only outperforms RT on all the cases but is also well-suited to generate statistical test data for fault-localization. Of course, more experiments are required to validate this statement on other programs but these results are clearly very encouraging.

6 Conclusion

In this paper, we proposed a new statistical testing method that generates sequences of random test data that respect the following probabilistic properties: 1) each sequence guarantees the uniform selection of feasible paths only and 2) the uniform selection of test data over the subdomain associated with these paths. We introduced a new algorithm that uniformly selects feasible paths only by drastically decreasing the number of rejects during the test data selection. This approach was implemented in a prototype for JAVA and preliminary experimental results were given to show that the generated test suite increases the size of DBBs w.r.t. a pure random testing approach. Moreover, we got encouraging experimental results that show the potential for increasing the effectiveness of a standard coverage-based localization fault technique. Future work will be dedicated to improve the coverage of the JAVA language, in particular by taking method calls into account. Moreover, additional experimental results are required to fully validate the ability of our statistical test data generator to generate test data for fault localization.

References

[1] R. Abreu, P. Zoeteweij, and A.J.C. van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference, Practice and Industrial Conference*, Windsor, UK, September 2007.

[2] B. Baudry, F. Fleurey, and Y. Le Traon. Improving test suites for efficient fault localization. In *Proceedings of International Conference on Software Engineering*, pages 20–28, Shanghai, China, May 2006.

[3] M. Carlsson, G. Ottoson, and B. Carlson. An open–ended finite domain constraint solver. In *Proceedings of the International Symposium on Programming Languages: Implementations Logics and Programs*, LNCS, pages 191–206, Soupthampton, UK, September 1997. Springer.

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

[5] R.A. DeMillo and J.A. Offutt. Experimental results from an automatic test case generator. *ACM Transactions on Software Engineering and Methodology*, 2(2):109–127, April 1993.

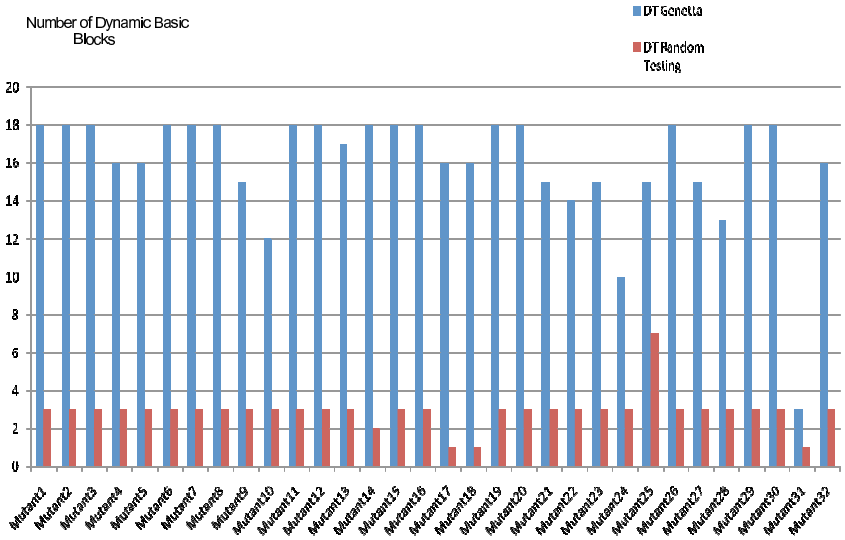

Figure 2. Comparison of the number of dynamic basic blocks for a Genetta and a Random Testing test suite

[6] A. Denise, M.-C. Gaudel, and S.-D. Gouraud. A generic method for statistical testing. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 25–34, St-Malo, France, November 2004. IEEE.

[7] J.W. Duran and S. Ntafos. An evaluation of random testing. *IEEE Transaction on Software Engineering*, 10(4):438–444, July 1984.

[8] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 213–223, Chicago, USA, June 2005. ACM.

[9] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 53–62, Clearwater Beach, USA, March 1998.

[10] A. Gotlieb and M. Petit. Constraint reasoning in path-oriented random testing. In *Proceedings of the International Computer Software and Applications Conference*, Turku, Finland, July 2008.

[11] P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(fd). Technical Report CS-93-02, Brown University, 1993.

[12] J.A. Jones, M.J. Harrold, and J. T. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the International Conference on Software Engineering*, pages 467–477, Orlando, USA, May 2002.

[13] P. L' Ecuyer. *Random Number Generation, Chapter 2 of Handbook of Computational Statistics: Concepts and Methods*. Springer-Verlag, 2004.

[14] J.D. Musa. Operational profiles in software-reliability engineering. *IEEE Software*, 10(2):14–32, March 1993.

[15] G. J. Myers. *The Art of Software Testing*. John Wiley, New York, 1979.

[16] M. Petit and A. Gotlieb. Boosting probabilistic choice operators. In *Proceedings of the International Conference on Principles and Pratice of Constraint Programming*, LNCS, pages 559–573, Providence, USA, September 2007.

[17] M. Petit and A. Gotlieb. Uniform selection of feasible paths as a stochastic constraint problem. In *Proceedings of the International Conference on Quality Software*, IEEE, pages 280– 285, Portland, USA, October 2007.

[18] P. Thévenod-Fosse and H. Waeselynck. An investigation of statistical software testing. *Software Testing, Verification and Reliability*, 1(2):5–25, July 1991.

[19] Y. Usui and S. Chiba. Bugdel: An aspect-oriented debugging system. In *Proceedings of Asia-Pacific Software Engineering Conference*, pages 790–795, Taipei, Taiwan, December 2005.

[20] N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *Proceedings of the European Dependable Computing Conference*, pages 281–292, Budapest, Hungary, April 2005. Springer.

[21] Y. Yu, J.A. Jones, and M.J. Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *Proceedings of the International Conference on Software Engineering*, pages 201–210, Leipzig, Germany, May 2008.

[22] M. Yu-Seng, J.A. Offut, and R. Kwon. Mujava : An automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133, June 2005.