# Probalistic Formula Testing

**Laurent Bouaziz,**
CERMICS – ENPC
Central 2 – La Courtine
F-93167 Noisy le Grand cedex

**Gilles Bernot, Pascale Le Gall**
Université d'Evry, LaMI
Cours Monseigneur Roméro
F-91025 Evry cedex
{bernot,legall}@lami.univ-evry.fr

## Introduction

In the field of software or hardware testing [1], the systematic approaches to generate test data sets are mostly based on a first step where the input domain is divided into a family of subdomains [2, 3]. According to the so-called *structural*, or *white-box*, testing approach, these subdomains correspond to well chosen paths in the data flow or control graph of the product under test. According to the so-called *functional*, or *black-box*, testing, these subdomains correspond to the various cases addressed by the specification of the product under test. Then, there are two main strategies to select test cases :

- either one performs a *deterministic* selection of test cases within each subdomain;
- or one performs a *probabilistic* selection of test cases based on a distribution on each subdomain [4].

If the test data set is successfully executed, it provides us with different kinds of confidence :

- If the selection has been deterministic, then we get a *qualitative* reliability evaluation, namely the confidence that we give to the criteria used to make the partition of the input domain into subdomains (e.g., to cover all the branches of the control graph, or to cover all the i-paths [5]).
- If the selection has been probabilistic, then we can get a *quantitative* reliability evaluation, given by classical probability results, according to chosen distributions (e.g., using the operational profile [6]).

*Deterministic structural* testing has already been extensively studied [1] and several test generation tools exist. There are ongoing researches about *probabilistic structural* testing (e.g., [7, 2, 3]) with promising concrete results [8].

*Deterministic functional* testing is also widely practiced (the test cases are established while writing the specifications). With the emergence of formal specifications, several research works show that it becomes possible to define the functional testing strategies in a rigorous and formal framework, and to partly automate them [9, 10]. All these approaches are deterministic ones.

The work reported here aims at rigorously treating *probabilistic functional* testing. Let us make precise that we address *dynamic* testing, i.e., the generated test cases are *executed* by the system under test and we check the actual results against the ones defined by the specification.

A formal specification of a functionality $f$ can be seen as a (possibly composed) formula establishing the required properties between the input variables of $f$, say $x_1, x_2, \cdots, x_n$, and the output result $f(x_1, x_2, \cdots, x_n)$. Thus, we can consider that we test formulas of the form: $\forall\,(x_1, \cdots, x_n) \in D\ ,\ \varphi(x_1, \cdots, x_n)$
In practice, to test such formulas amounts to :

- generate a (pertinent) test data set, which is a finite subset of the input domain $D$ of the formula, i.e., some chosen values for the tuple $(x_1, \cdots, x_n)$,
- execute each test, which means make the product under test execute the terms of $\varphi$ for the chosen values,

- if one of the execution does not furnish the expected results (i.e., they do not logically satisfy $\varphi$), then the test reveals a failure, else all the tests are in success and it remains to evaluate our confidence into the correctness of the product under test with respect to the formula.

Such a process requires a lot of instrumentations. For instance in the last step, which consists in deciding the success/failure of each test, it is necessary to be able to decide if $\varphi(x_1, \cdots, x_n)$ is satisfied. Thus, it is necessary to make available a *decidable* "oracle" ([11]) which computes $\varphi$ for any $(x_1, \cdots, x_n)$ in the domain. In this article we only focus on the first step, assuming that the second and last steps are already instrumented.

The next section addresses the question at the theoretical level, the "formula testing" level. We mainly address three questions. The first one is "how many test cases should we generate in order to affirm that the system will behave correctly, except possibly for a given percentage of the input values?" The second question is "how to evaluate the risk that we take when we guaranty this percentage, and sign the permit to deliver the system?" The last question is "how to choose pertinent test cases?" It is the reason why we introduce the notion of $(\mu, \epsilon, \alpha)$-*validation*.

- $\mu$ is a distribution on the domain $D$ of the formula. Roughly speaking, $\mu$ specifies a way to generate *well chosen* test cases out of the domain, and gives a precise meaning to the expression "well chosen."
- $\epsilon$ can be seen as a contract between the vendor of the product under test and the client. It allows the vendor to say "according to $\mu$, I affirm that my product satisfies the formula $\varphi$, with a probability $\epsilon$ to be wrong."
- $\alpha$ can be seen as the risk that the vendor takes when making this affirmation.

According to this theory, we propose a tool to assist test generation. The point is to generate test cases from a description of the domain $D$. We have considered several primitive operations to describe the most common domains in computer science. These operations on domains are treated and a small prototype written in MATHEMATICA [12] allows to assist test generation on domains definable according to these primitive operations. Thus, the "complicated underlying probabilistic machinery" is hidden behind a "set theoretic interface."

Some related works are considered in the last section. We show that our approach is not limited to probabilistic *functional* testing only. A triple $(\mu, \epsilon, \alpha)$ can also be deduced from probabilistic *structural* testing, and combined with the functional one, in order to better estimate and tune the risk taken by the vendor. Lastly, we outline some possible cross-fertilization between our approach and a recent tool performing deterministic functional test generation from algebraic specifications [17].

# 1 Formula Testing

## 1.1 $(\mu, \epsilon, \alpha)$-validation

The two propositions below are the basic results on which we rely to replace an exhaustive deterministic verification of a formula with a finite probabilistic verification. Let us first introduce some notations.

- The tested formula is of the form $\forall\ X\ \in\ D\ ,\ \ \varphi(X)$ where $X$ is a variable which replaces the tuple $(x_1, \cdots, x_n)$ of the formulas stated in the introduction.
- The domain $D$ of the input variable $X$ of the formula is assumed to be countable.
- $\mu$ is a probability distribution on $D$ that gives a strictly positive weight to any element of $D$: such a distribution is termed *complete* on $D$.
- $(X_i)_{i\geq 0}$ are independent random variables on $D$ distributed according to $\mu$: this means that they are drawn at random[1], their result being distributed according to $\mu$ and the output of any subset of them does not influence the rest of the outputs.
- $F$ is the subset of $D$ for which $\varphi$ does not hold, i.e.: $F = \{X \in D \mid \neg\varphi(X)\}$. Our goal is to propose a validation procedure to check whether $F$ is empty.

[1] for simplicity reasons, we do not distinguish between the random variable, which is a mapping, and its realization, which is an element of $D$.

**Proposition 1** $(\forall\ X \in D,\ \varphi(X)) \quad \Leftrightarrow \quad (\forall\ i \geq 0,\ \varphi(X_i))$

(To be fully correct within a probabilistic framework, the above assertion holds up to the almost sure equivalence. This has no practical consequence.) This result alone would be of no practical value: we simply replace a countable deterministic verification with a countable random check. The next proposition shows that in the latter case, it is possible to infer nonetheless from a finite *probabilistic* verification a quantitative estimate of the confidence we can have in the formula. Let us first introduce a definition to make things more precise (see also Section 1.2):

**Definition 1** *A $\mu$-test of length $N$ is any set $\{\varphi(X_1), \ldots, \varphi(X_N)\}$. Such a test is said* in success *if for all $i = 1..N$, $X_i \not\in F$ (i.e., $\varphi(X_i)$ holds).*

In analogy to what happens in statistical quality control where one admits the possibility of wrongly accepting a decision, we introduce the following notion of probabilistic validation:

**Definition 2** *We call* $(\mu, \epsilon, \alpha)$-validation *of a formula $\varphi$ any procedure that allows one to assess:*

$$\textit{With a probability of at most } \alpha \textit{ to err, } \mu(F) \leq \epsilon$$

The error $\alpha$ being considered here is what statisticians call "error of the second kind," i.e., the error that one makes when one announces that the result holds when it does not.

**Proposition 2** *If a $\mu$-test of length $N$ succeeds, it is a $(\mu, 1 - \sqrt[N]{\alpha}, \alpha)$-validation of $\varphi$.*

Let us notice that as $N$ grows and assuming that the tests are successful, we can give estimates on the upper bound for the probability of $F$ that gets closer to 0, which is rather logical.
$\alpha$, that lies between 0 and 1, measures the quality of the test: the closer $\alpha$ is to 0, the greater the confidence in the validation. In other terms, if one repeats 100 times the previous test (with independent draws), the decision will be wrong in at most $100\,\alpha$ cases.

### 1.2 Formula Testing Applied to Probabilistic Functional Testing

We want to deal with the test of a formula $(\forall\ X \in D,\ \varphi(X))$ , that comes from a specification. The previous results suggest the following approach:

1. Select a distribution $\mu$ on $D$.
2. Select a confidence level $1 - \alpha$ and a control parameter $\epsilon$.
3. Compute the length $N$ of the $\mu$-test according to: $N \geq \frac{\log(\alpha)}{\log(1-\epsilon)}$
4. Draw $N$ times in the distribution $\mu$ and for each of the produced values, compute the truth of $\varphi$.
5. If the previous $\mu$-test of length $N$ succeeds, then we have a $(\mu, \epsilon, \alpha)$-validation of $\varphi$.

The implementation of the approach assumes that one is able to draw at random according to a given distribution. This is the problem we are going to tackle now.

## 2 Random Generation

### 2.1 An Introductory Example

Let us consider a function `intdiv` that computes the quotient $q$ of the division of two natural numbers $a$ and $b$: $\texttt{intdiv} : [0, MaxInt] \times [1, MaxInt] \rightarrow [0, MaxInt]$. A required property for `intdiv` is:
$\forall (a, b) \in [0, MaxInt] \times [1, MaxInt]\ , \quad (0 \leq a - b \times \texttt{intdiv}(a, b)) \quad \& \quad (a - b \times \texttt{intdiv}(a, b) \leq b - 1)$
The operations $-$, $\times$, $\leq$, & and of course `intdiv` are assumed to be executable.

An *elementary* test of the previous formula amounts to select two values $a_0$ and $b_0$ and to submit the formula to the program. To benefit from the approach we developed in the previous section, we have to draw according to a distribution on $[0, MaxInt] \times [1, MaxInt]$. In the present case, this is quite obvious since we can choose *for example* the uniform distribution on that set that assigns a weight of $\frac{1}{(MaxInt+1)MaxInt}$ to any pair of the input domain.

The choice of the uniform distribution is not imposed by the method however. The tester may prefer to give a special importance to values near to the boundary of the input domain. This could lead, for example, to a procedure that would draw: with probability $1/2$ a pair in the set $[0, MaxInt] \times \{MaxInt\}$, and with probability $1/2$ a pair in the complementary set.

## 2.2 Goal

According to the theory developed previously, the distribution in which data are drawn must assign a non-zero weight to any element of the input domain $D$, otherwise, the validation would hold only for the carrier set of the distribution. Hence:

**Definition 3** *A* generation function *for a domain $D$ is a procedure (in the computer science meaning) that outputs values according to a* complete *distribution.*

Our goal is now to produce generation functions for a large class of sets frequently used in computer science. We are going to list and comment the basic blocks and the combinators used to describe sets. For each operation, we will provide both a short mathematical justification of the method and some examples of the possible uses of the tool. All the examples and the code are written in MATHEMATICA ([12]).

**Definition 4** *A simulation pair is a pair $(D, \gamma)$ where $D$ is a set and $\gamma$ is a generation function on $D$.*

The following subsections define inductively the class $\mathcal{S}$ of simulation pairs handled by our tool.

## 2.3 Interval of Integers

The first class of basic sets we consider is the class of the interval of integers denoted by `intInterval[{a,b}]`. The tool allows to build either the uniform distribution or any specified distribution defined by the weights assigned to each element of the interval. By default, `generate[intInterval[{a,b}]]` draws numbers in the interval `[a,b]` according to the uniform distribution. To implement such kind of generator, we make the classical assumption ([13]) that we can rely on a perfect generator that simulates a uniform distribution on $[0, 1]$ as a subset of the reals. `generate[intInterval[{a,b},d]]`, where `d` is a density function that assigns a probability to each element of `[a,b]`, draws numbers in `[a,b]` according to `d`. For example, we can set:

```
d[0]:=1/3 ; d[1]:=2/3 ; int01:=intInterval[{0,1},d]
```

and in average, 1 will be drawn twice as much as 0. We can also consider a uniform distribution on bounded natural numbers: `bnat:=intInterval[{0,MaxInt}]`

## 2.4 Enumerated Set

Since finite sets can be seen as mapped integer intervals, their generation is straightforward; for example, with the previous `d`, we can generate a boolean with: `bool:=finiteSet[{false,true},d]`
Moreover, for practical purposes, we introduce a `Singleton` operation with an obvious meaning.

## 2.5 Cartesian Product

Given any tuple of simulation pairs $(D_i, \gamma_i)$, it is easy to build the simulation pair $(\prod_i D_i, \gamma)$ where $\gamma$ is the tuple whose $i$-th component is equal to $\gamma_i$. The probability to draw a given tuple $(u_1, \cdots, u_i, \cdots)$ is equal to the product of the probabilities to draw each $u_i$ according to the distributions of the $\gamma_i$. For example:

```
fprod:=product[int01,bool]
```

`generate[fprod]` will return pairs $(i, b)$ where $i$ is drawn in $[0, 1]$ according to `d` and $b$ is a boolean drawn according to `d` too.

## 2.6 Union

Given a tuple of simulation pairs $(D_i, \gamma_i)_{1 \leq i \leq n}$ and a family of strictly positive weights $(w_i)_{1 \leq i \leq n}$, it is possible to build a generation function on $\cup_i D_i$: one has to draw an index $i_0$ on the interval $[1, n]$ according to the distribution given by the $w_i$ and then draw in $D_{i_0}$. If the carrier sets $D_i$ are disjoint, then the $w_i$ can be interpreted as the relative frequency with which values will be drawn in the $D_i$. The probability to draw in $\cup_i D_i$ an element $u$ of a given $D_j$ is equal to the product of $\frac{w_j}{\sum_i w_i}$ by the probability to draw $u$ in $D_j$ according to $\gamma_j$. For example, one may have defined:

```
int01prime:=Union[{{Singleton[0],1/3} , {Singleton[1],2/3}}]
```

and the distributions associated to `int01` and `int01prime` would have been identical.

## 2.7 Mapping

Given any simulation pair $(D, \gamma)$ and any function `g` whose domain coincides with $D$, it is possible to build a generation function on the codomain of `g` by simply first drawing a value in $D$ and then mapping it with `g`. The probability of an element $u$ of `g(`$D$`)` is the sum of the probabilities of the antecedents of $u$ by `g`. One may have thus defined:

```
g[0]:=false ; g[1]:=true ; boolprime:=map[int01,g]
```

and the distributions associated to `bool` and `boolprime` would have been identical.

## 2.8 Countable Set

Given any family of positive real numbers $w_i$ that sum up to 1, it is possible to build a distribution on the set of *natural numbers*, denoted `nat` that assigns a probability $w_i$ to $i$. That is the goal of the primitive `nat[w]` where `w` is the weight function. By default, the following distribution is assigned to `nat`:

```
nat:=Union[{{intInterval[{0,MaxInt}],1-EPS},{intInterval[{MaxInt+1,INFINITY}],EPS}}]
```

Any interval of the form `intInterval[{x,INFINITY}]` is provided by default with a generation function `poisson` which is the density of a Poisson distribution whose intensity is 1:

```
poisson[n,lambda,x]:=Exp[-lambda] lambda^(n-x+1)/Factorial[n-x+1]
```

The intuition behind `nat` is to have a uniform distribution up to a prespecified number $MaxInt$ mixed with a rapidly decreasing distribution for numbers greater than $MaxInt$, the weight attributed to this last distribution being controlled by a constant `EPS`. By mapping, this allows to build a distribution on any countable set that is defined as the range of some given map on the set of natural numbers.

## 2.9 Subset

It is often convenient to define a set $D'$ as the subset of a bigger one $D$ through a predicate $p$. If this predicate is executable and if we have a generation function on $D$, $\gamma$, the rejection method ([13]), which amounts to draw in $D$ as long as the predicate is not satisfied, gives a general method to build a generation function on $D'$. The greater the probability of $D'$ under the distribution associated with $\gamma$, the shorted the average time needed to draw in $D'$. Given that the function `IsPrime` checks that a given natural number is prime, it is easy to draw prime numbers: `prime:=subset[nat,IsPrime]`. The efficiency of the previous generation function is not so bad since the asymptotical density of prime numbers is $\frac{\log(n)}{n}$.

## 2.10 Sequence

One has often to deal with "product" sets where one factor of the product depends on some other factors. For example, $\left\{(x,y) \in bnat^2 |\ x \leq y\right\}$ is such a set. It is often more efficient to express explicitly this dependency than to consider such a set as the subset of a bigger one. That is why we provide a `Sequence` operation that allows us to describe the previous set as:

```
DBNat := Sequence[{bnat,Function[x,intInterval[x,MaxInt]]}]
```

and this amounts to first draw $x_0$ in `bnat` and then to draw uniformly in $[x_0, MaxInt]$ and the probability to draw $(x,y)$ is $\frac{1}{(MaxInt+1)(MaxInt-x+1)}$.

The tool allows in fact to deal with any such "dependent product set" (corresponding to the classical dependent types in computer science) where one can find a permutation of the components such that in the reordered tuple, the $i$-th component depends only on the $1 \ldots (i-1)$-components.

## 2.11 Recursive Structures

Inductively defined sets are omnipresent in computer science. In order to keep things simple and short in this article, we will only deal with "linear" recursive structures like lists (the interested reader can consult [14] for a more general presentation).

### 2.11.1 Free Linear Recursive Structures

They can be seen as some least fixpoint for some building *total* functional. In the formal specification setup, it is convenient to work with the set of all the terms built on the signature of a given data type, and the functional is then called a *free constructor*.

The generation of free structures is very simple. For example, if one wants to generate lists of integers at random, one just has to draw first the length $l$ of the list, and then iteratively $l$ times, draw an integer and apply the free constructor "`cons`" to get a list. More precisely, given the following signature $\Sigma$:

```
op nil : -> List
op cons _ _ : Nat List -> List
op head _ : List -> Nat
op tail _ : List -> List
```

lists can be defined as the set of all the terms generated over `nil` by the constructor `cons`. Their general form is: `cons(x1,cons(x2,...(cons(xn,nil))...))`. This can be expressed as the least subset of all the $\Sigma$-terms satisfying: $X = \{\texttt{nil}\} \cup \texttt{cons}(nat, X)$ where $nat$ denotes the carrier set containing all the values of type `Nat` (previously provided with its own generation function).

```
list:=RecStruct[Sig->Sigma,Base->Singleton[nil],Cons->{cons}]
```

where `Sigma` is defined in an obvious way.

By default, all the lengths[2] up to `MaxInt` are considered equivalent and the other lengths are neglected. This is implemented via a default function `NDistrib` which draws the length $l$ according to the `nat` distribution. If we want to privilege short lists over longer lists, we can modify the distribution that controls the length of the generated structure with:

```
shortList:=ModifyDistrib[list,NDistrib->Union[{{intInterval[{0,SmallInt}],1/2},
                                               {intInterval[{SmallInt+1,INFINITY}],1/2}}]]
```

To generate non-empty lists, there are several possibilities:

- one first solution is to consider them as a subset of the lists and to define them as:
  `NeList1:=subset[list,Function[x,Not[IsEmpty[x]]]]`
- a second solution is to modify the distribution on the lengths to exclude a length of 0.
  `NeList2:=ModifyDistrib[list,NDistrib->intInterval[{x,INFINITY}]]`
- a third solution is to proceed from the ground up and to set as the base set the lists of length 1:
  ```
  NeList3:=RecStruct[Sig->Sigma,
                     Base->map[nat,Function[x,cons[x,nil]]],
                     Cons->{cons}]
  ```

### 2.11.2 Constrained Linear Recursive Structures

The main difference with the previous case is that the building functional can be partial. The domain of the functional is often defined by a predicate. The algorithm we provide here works if this predicate is defined in terms of a recursive function on whose range a generation function is available.

Let us take the example of non empty sorted lists *SortedList*. The building functional can be written as:

$$\forall (x,e) \in list \times nat, \begin{cases} x \in SortedList \ \& \ e \leq head(x) \Rightarrow cons(e,x) \in SortedList \\ cons(e,nil) \in SortedList \end{cases}$$

Let us notice that the constraint is expressed in terms of the recursive function *head* whose range is *nat*: the property we required from the predicate is thus satisfied.
The basic idea of the generation algorithm is then to try to solve a problem of the form: $\{ head\ (x_0) = e_0$ $size(x_0) \leq n$, where *size* denotes the number of constructor calls needed to build $x_0$ and $n$ is a natural number. $n$ controls the size of the generated structure and will be drawn at random in order to generate structures of different sizes.

The next idea is to write $x_0$ under the form: $x_0 = cons(e_1, x_1)$. $x_0$ can be in *SortedList* if and only if $x_1$ is in *SortedList* and $e_1 \leq head(x_1)$. Moreover we must have $head(x_0) = e_0$. Because of the axiom relative to *head* (see the next section), we have: $\{ e_1 \leq head(x_1)$
$e_0 = e_1$
Let us consider the subset $G_{u,n}$ of $nat \times nat$ defined by: $\{ v_1 \leq v_2$
$u = v_1$. This set can be expressed as:

```
G[u_,n_]:=Sequence[{Singleton[u],Function[x,intInterval[{x,INFINITY}]]}]
```

Once a pair $(e_1, e_2)$ has been drawn in $G_{u,n}$, we have to solve the following problem: $\{ head\ (x_1) = e_2$ $size(x_1) \leq n-1$, which leads to a straightforward recursive solution of the generation problem.
The recursion stops when:

- either $G_{u,n} = \emptyset$ (which never happens here as can be seen from its expression above) and with the previous notations, $x_1$ is drawn in $B$.
- or $n = 0$ and once again, $x_1$ is drawn in $B$.

The distribution that results from the above algorithm is naturally parameterizable through the choice of a distribution: on the size of structure, on the set $G_{u,n}$, and on the range of the recursive function for which the equation is solved (*head* here).

Finally, to test a formula of the form $\forall X \in D, \varphi(X)$, it suffices to produce a set description of $D$, based on the primitives shown above.

[2] the length being defined here as the number of constructors.

# 3 Relationship with some Other Approaches

## 3.1 Partition Testing

Partition testing ([15, 2, 16, 3]) is a classical testing method which consists in breaking the input domain $D$ in several pieces and in drawing *uniformly* in each subdomain $D_i$ a given number of data $n_i$. The result of such a test can be expressed in terms of a $(\mu, \epsilon, \alpha)$-validation. More precisely, let us denote: $w_i = \frac{n_i}{\sum_j n_j}$ , $\bar{\mu} = \sum_i w_i \mu_i$ and $\bar{n} = \sum_i n_i$ where $\mu_i$ is the uniform distribution on $D_i$. Then, we have the following result:

**Proposition 3** *The success of the previous partition test is a* $(\bar{\mu}, \sum_i w_i(1 - \sqrt[n_i]{\alpha}), 1-(1-\alpha)^{\bar{n}})$*-validation.*

## 3.2 Statistical Structural Testing

Statistical structural testing [7, 8] consists also in breaking the input domain into subdomains $D_i$. Rather than choosing arbitrary values in each subdomain, the authors of the method suggest to draw at random on the whole domain, arguing that this compensates for the imperfection of any prespecified criterion. More formally, the distribution $\mu$ is constructed in such a way that for each subdomain $D_i$, the probability that one value is drawn in $D_i$ must be greater than a prespecified quantity $q$ termed *the test quality*.

A straightforward computation shows that the length $n$ of such a test must verify the following relation: $n \geq \max_i \frac{\log(1-q)}{\log(1-\mu(D_i))}$ . The next proposition follows immediately from this observation:

**Proposition 4** *Given a successful statistical structural test with distribution* $\mu$*, partition* $D_i$ *and test quality* $q$*, it is possible to deduce, for any given* $\alpha$ *in* $]0, 1[$*, a* $(\mu, 1 - \sqrt[\bar{n}]{\alpha}, \alpha)$*-validation, with:* $\bar{n} = \max_i \lceil \frac{\log(1-q)}{\log(1-\mu(D_i))} \rceil$

Let us finally notice that the generation tool described previously helps one to build the functions needed by a partition test as well as by a statistical structural test.

## 3.3 Deterministic Functional Testing

Once a successful test has been conducted, if we want to increase the reliability evaluation, we get two different scenarios depending on whether the test data selection is deterministic or probabilistic:

- Deterministic: we can refine the criteria, and it will result into numerous smaller subdomains in which a few test cases are selected.
- Probabilistic: we can either tune the distribution in order to privilege some special cases or increase the number of generated test cases in order to get a more interesting triple $(\mu, \epsilon, \alpha)$.

The first scenario presents the advantage that the new subdomains exhibit cases addressed either by the program or by the specification that are likely to reveal errors. However, it cannot always be done automatically. The second scenario presents the advantage that the test has a length under control and can be automatically generated provided that the generation function is available. In return, there is no guarantee of quickly revealing some prespecified test cases (in relation to the structure of either the system or the specification).

In the field of functional testing, B. Marre has developed LOFT, a tool for deterministic test data selection from classical positive conditional algebraic specifications [17]. It is written in PROLOG and is based on an equational resolution procedure with some control mechanisms. The main mechanism for defining subdomains is the decomposition based on a case analysis. This case analysis is achieved by unfolding the validity domain of the axioms w.r.t. the structure of the specification into a partition of smaller validity subdomains. It would be interesting to combine LOFT with our tool of probabilistic test

generation in order to benefit from the fine decomposition into small subdomain provided by LOFT and from our quantitative reliability evaluation. The main difficulty comes from the fact that the subdomains given by LOFT are characterized by predicates and thus do not necessarily belong to our class of $\mathcal{S}$ of simulations pairs. Such a combination of the two tools requires to find an intermediate level where subdomains can be described both by a predicate (as in LOFT) and by using building primitives (as in our approach).

# 4 Conclusion

We have defined a framework for *probabilistic functional testing*. Our main contribution is the formalization of the testing activity in term of $(\mu, \epsilon, \alpha)$-validation. It allows to associate to any successful test of length $N$ drawn according to the distribution $\mu$ two useful quantitative measures: $\epsilon$ which gives a probabilistic upper bound of the potential error domain and $\alpha$ which gives a clue to help the tester/vendor to estimate the risk (s)he takes in underestimating the measure of the error domain. These two measures give a quantitative evaluation of the reliability. In order to facilitate the use of our theory, we have proposed a tool to assist test generation on the most common domains in computer science (as Cartesian product, inductively defined set, ...). Our tool is only a first prototype which proves the applicability of our method. In order to fully illustrate the interest of our method, one should compare or combine it with other approaches on some real sized case studies. Of course, for this, our tool should offer a more user-friendly interface.

# References


[1] B. Beizer: *Software testing techniques.* Van Nostrand Reinhold, New-York, Second edition. 1990.

[2] E.J. Weyuker, B. Jeng: *Analysing partition testing strategies.* IEEE Trans. Software Engineering, Vol.17, No.7, p.703-711, July. 1991.

[3] T.Y. Chen, Y.T. Yu: *On the expected number of failures detected by subdomain testing and random testing.* IEEE Trans. on Software Engineering, Vol.22, No.2, p.109-119, February. 1996.

[4] J.W. Duran, S.C. Ntafos: *An evaluation of random testing.* IEEE Trans. on Software Engineering, Vol.10, p.438-444, July. 1984.

[5] R. Hamlet: *Theoretical comparison of testing methods.* Proc. of the 3rd Symposium on Software Testing, Analysis and Verification (TAV-3), Key West, USA, Software Engineering Notes, Vol.14, No.8, p.28-37, December. 1989.

[6] M. Dyer: *The cleanroom approach to quality software development.* John Wiley and sons. 1992.

[7] P. Thevenod, H. Waeselynck, Y. Crouset: *An experimental study on software structural testing: determinisitc versus random input generation.* Proc of the 21st IEEE Symposium on Fault-Tolerant Computing, Montreal, p.410-417. 1991.

[8] B. Marre, P. Thévenod, H. Waeselynk, P. Le Gall, Y. Crouset: *An experimental evaluation of formal testing and statistical testing.* Proc. of Safety of Computer Control System 1992 (SAFECOMP'92), Zurich, October 1992, IFAC (Heinz H. Frey Ed.), Pergamon Press, p.311-316. 1992.

[9] J. Dick, A. Faivre: *Automating the generation and sequencing of test cases from model-based specifications.* Proc. of Formal Methods Europe (FME 93), Springer-Verlag LNCS 670, p.268-284. 1993.

[10] A. Arnould, P. Le Gall, B. Marre: *Dynamic testing from bounded data type specifications.* Proc. of EDCC-2, Second European dependable Computing Conference, Taormina, Italy. 1996.

[11] E.J. Weyuker: *On testing non testable programs.* The Computer Journal Vol.25, No.4, p.465-470. 1982.

[12] S. Wolfram: *Mathematica: A System for doing Mathematics.* Addison Wesley. 1995.

[13] L. Devroye: *Non-uniform random variate generation.* New-York, Springer. 1986.

[14] L. Bouaziz: *Méthodes probabilistes pour la validation de formules et applications au test de logiciel.* Thèse de Doctorat, Ecole Nationale des Ponts et Chaussées, Paris. 1996.

[15] R. Hamlet, R. Taylor: *Partition testing does not inspire confidence.* IEEE Trans. on Software Engineering, Vol.16, p.1402-1411, December. 1990.

[16] M.Z. Tsoukalas, J.W. Duran, S.C. Ntafos: *On some reliability estimation problems in random and partition testing.* IEEE Transactions on software Engineering, July. 1993.

[17] B. Marre: *Toward automatic test data set selection using algebraic specifications and logic programming.* Proc. of the 8th Intl. Conference on Logic Programming (ICLP'91), Paris, June 1991, Logic Programming M.I.T. Press, p.202-219. 1991.