A Theory of Probabilistic Functional Testing

Gilles Bernot
Université d'Evry
LaMI
Cours Monseigneur Roméro
F-91025 Evry cedex, France
bernot@lami.univ-evry.fr

Laurent Bouaziz
CERMICS – ENPC
Central 2 – La Courtine
F-93167 Noisy le Grand cedex,
France
bouaziz@cermics.enpc.fr

Pascale Le Gall
Université d'Evry
LaMI
Cours Monseigneur Roméro
F-91025 Evry cedex, France
legall@lami.univ-evry.fr

ABSTRACT
We propose a framework for "probabilistic functional testing." The success of a test data set generated according to our method guarantees a certain level of confidence into the correctness of the system under test, as a function of two parameters. One is an estimate of the reliability, and the other is an estimate of the risk that the vendor takes when (s)he notifies this reliability percentage to the client. These results are based on the theory of "formula testing" developed in the article. We also present a first prototype of a tool which assists test case generation according to this theory. Lastly, we illustrate our method on a small formal specification.

Keywords
software testing, random testing, formal specification, functional testing, partition testing, reliability, probabilistic testing.

INTRODUCTION

In the field of software or hardware testing [5, 2], 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 [28, 4]. According to the *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 *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 (for example one "nominal" case and some others "to the limits");
- or one performs a *probabilistic* selection of test cases based on a distribution on each subdomain [10] (for example the uniform distribution on the domain).

Assuming that, after submission, the test data set is successfully executed, it provides us with different kinds of confidence about the product under test :

- 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 (e.g., to cover all the branches of the control graph [14]).
- 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 as in the cleanroom approach [11]). A good motivation for this approach is to deal with the lack of infallible criteria and fault model.

Deterministic structural testing has already been extensively studied [5, 2] and, based on this approach, several (not only academic) test generation tools exist. There are ongoing researches about *probabilistic structural* testing (e.g., [25, 28, 4]) with promising concrete results [21].

Deterministic functional testing is also widely practiced (the test cases are established while writing the specifications). As far as informal specifications are used, this process is mainly performed manually, or by manually extracting formal properties or graphs from the informal specifications and then apply systematic processes. With the emergence of formal specifications such as VDM [17], Z [23], or algebraic specifications (e.g., LARCH [13] or OBJ [18]), 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 [8, 24, 9, 16, 12, 1]. All these approaches are deterministic ones, they do not address the probabilistic side.

The work reported here aims at rigorously treating *probabilistic functional* testing. Functional testing approaches have the interesting property to facilitate the prevision of the expected result for each test case (the role of a specification being precisely to characterize

the acceptable results). Besides, functional testing approaches are complementary to structural ones, as they allow to test if all cases mentioned in the specification are actually dealt with in the product under test, which is not the case of structural testing (if a case has been forgotten in a program, the corresponding path is missing, thus it is probably not exercised). Moreover, only probabilistic methods can ensure valuable *quantitative* reliability estimates. Intuitively, this results from the guaranty that any sample generated according to a given distribution provides informations on further samples that would be drawn according to the same distribution. Moreover, an arbitrary quality level being given, probabilistic methods are able to furnish the size of the sample to be drawn.

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, as opposed to *static* testing, which rely on some source checking [11].

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 denoted by $f(x_1, x_2, \cdots, x_n)$. The input tuple $(x_1, x_2, \cdots, x_n)$ has to belong to the intended domain D_f of the functionality f. Consequently, a formal specification of f is something of the form

$$\forall\ (x_1, x_2, \cdots, x_n)\ \in\ D_f\ ,\\ \psi(x_1, x_2, \cdots, x_n, f(x_1, x_2, \cdots, x_n))$$

where ψ is the required input/output property. More generally, we can consider that we test formulas of the form:

$$\forall\ X\ \in\ D\ ,\ \varphi(X)$$

where to simplify notations, X is a variable which replaces the tuple $(x_1, \cdots, x_n)$ and $\varphi(X)$ plays the role of the formula $\psi(x_1, x_2, \cdots, x_n, f(x_1, x_2, \cdots, x_n))$.

In practice, to test such formulas amounts to :

- generate a (pertinent) test data set, which is, consequently, a finite subset of the input domain D of the formula, i.e., some chosen values for the tuple $X = (x_1, \cdots, x_n)$,
- execute each test, which means make the product under test execute $f(x_1, \cdots, x_n)$ for each previously chosen value,
- if one of the execution does not furnish the expected result, 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.

Of course, 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 indeed necessary to be able to decide if $\varphi(X)$ is satisfied. Thus, it is necessary to make available a *decidable* "oracle" ([27]) which computes φ for any $X = (x_1, \cdots, x_n)$ in the domain. Similarly, executing f (in the second step) may require some instrumentations (e.g., to simulate its environment in order to introduce the input variables x_i). 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 purely theoretical level, the "formula testing" level. We adapt and formulate some classical results from probability/statistics and from software reliability engineering to our framework. To some extent, dynamic testing can be shown as the particular case of the software reliability area ([22]) where no error is discovered, and no software modification is done. It corresponds to a kind of "static" software reliability engineering where the notion of time is avoided (we do not address software reliability models, MTBF, etc.). We take benefit of this specialization in order to fully control the submitted tests data sets. We provide test generation methods and tools which ensure a statistical notion of complete coverage. More precisely, in any process of probabilistic software or hardware verification and validation, 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, but certainly not least, question is "how to choose pertinent test cases?" It is the reason why we introduce the notion of (μ, ϵ, α)-*validation*.

- μ is a distribution on the domain D of the formula. Roughly speaking, μ specifies a way to generate *well chosen* test cases out of the domain, and gives a precise meaning to the expression "well chosen" (it may be for example the well known uniform distribution, or the also well known operational profile).
- ϵ 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 μ, I affirm that my product satisfies the formula φ, with a probability ϵ to be wrong."
- α can be seen as the risk that the vendor takes when making this affirmation. Let us assume that N tests have been generated and that they are successful during the vendor verification, then, in average, if 100 clients generate their own N tests according to μ, at most $\alpha \times 100$ of them will find a bug.

According to this theory, we propose a tool to assist test generation, a first prototype of which is described in the section about "random generation." The point is to gen-

erate test cases from a description of the domain D. We have considered several primitive operations to describe the most common domains in computer science. For example, obviously, Cartesian product of domains, union of domains and so on are such useful operations, and, more importantly, recursively defined domains are especially useful in computer science. All these operations on domains are treated and a small prototype written in MATHEMATICA [29] 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." Since software engineers usually know simple set theory and recursivity, our tool is clearly useful.

An example of test generation from a formal specification, more precisely an algebraic specification with subsorts (OBJ [18]), is given in the section called "an example." The specification (*sorted lists*) is easily readable even without knowing the OBJ language, and a probabilistic test case generation in the sorted list domain demonstrates how our tool can handle dependent types.

Some related works are considered in the last section. We show that our approach is not limited to probabilistic *functional* testing only. A triple (μ, ϵ, α) 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 [20].

FORMULA TESTING

(μ, ϵ, α)-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 that will prevail for the rest of the paper.

- The tested formula is of the form
$$\forall\ X\ \in\ D\ ,\ \varphi(X)$$
- The domain D of the input variable X of the formula is assumed to be countable.
- μ 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 μ: this means that they are drawn at random[1], their result being distributed according to μ and the output of any subset of them does not influence the rest of the outputs.

[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.

- F is the subset of D for which φ 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.

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

Remark

- To be fully correct within a probabilistic framework, the above assertion holds up to the almost sure equivalence. This has no practical consequence.
- The key point in the proof of Proposition 1 is the condition that μ gives a *strictly* positive weight to any element of D.

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 the next subsection):

Definition 1 *A μ-test of length N is any set $\{\varphi(X_1), \ldots, \varphi(X_N)\}$. Such a test is said* in success *if for all $i = 1..N$, $X_i \notin F$ (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* (μ, ϵ, α)-validation *of a formula φ any procedure that allows one to assess the following:*

With a probability of at most α to err,
$$\mu(F) \leq \epsilon$$

Remark The error α 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 *Let us assume that a μ-test of length N succeeds. Then, it is a $(\mu, 1 - \sqrt[N]{\alpha}, \alpha)$-validation of φ.*

Remark

- This result is a consequence of classical statistical results [6].

- 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.
- α, that lies between 0 and 1, measures the quality of the test: the closer α 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.

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 (an axiom or a logical consequence of the specification). Under the assumption that the truth value of the formula is directly computable for any instance of the input data, the previous results suggest the following approach:

1. Select a distribution μ on D.
2. Select a confidence level $1-\alpha$ and a control parameter ϵ.
3. Compute the length N of the μ-test according to:
$$N \geq \frac{\log(\alpha)}{\log(1-\epsilon)}$$
4. Draw N times in the distribution μ and for each of the produced values, compute the truth of φ.
5. If the previous μ-test of length N succeeds, then we have a (μ, ϵ, α)-validation of φ.
6. Even if the previous test is not fully successful, it is nonetheless possible to infer from the number k of failures an estimation of $\mu(F)$. Easy statistical computations show that with a probability of at most α to err:
$$\mu(F) \leq \epsilon_0$$
where ϵ_0 is the maximum[2] of k/N and of the solution of the following equation:
$$\alpha = \left(\begin{array}{c} N \\ k \end{array} \right) \epsilon_0^k (1-\epsilon_0)^{N-k}$$
This kind of result still fits in our framework, since the previous estimate can be interpreted as a $(\mu, \epsilon_0, \alpha)$-validation.

The implementation of the approach assumes that one is able to draw at random according to a given distribution. Since we assumed that the truth decision for the formula can be automated, the only impediment to a full automatization of the testing procedure lies in this random generation phase. This is the problem we are going to tackle now.

RANDOM GENERATION

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:

$$\forall (a,b) \in [0, MaxInt] \times [1, MaxInt],$$
$$(0 \leq a - b \times \texttt{intdiv}(a,b))$$
$$\&$$
$$(a - b \times \texttt{intdiv}(a,b) \leq b - 1)$$

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 $[0, MaxInt] \times \{MaxInt\}$;
- with probability 1/2 a pair in the complementary set.

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.*

Let us notice that we do not impose any restriction on the kind of distribution except completeness:

1. the theory does not impose any such restriction, e.g. uniform distribution or the operational profile are only particular cases (see below).
2. we will take advantage of this flexibility in allowing testers to emphasize the subdomains they believe to be critical.
3. from a practical point of view, uniform distributions are not easy to produce automatically. In relaxing that constraint, we will be able to provide a tool that translates automatically set descriptions

[2] the function $x \rightarrow x^k(1-x)^{N-k}$ is decreasing on the interval $[\frac{k}{N}, 1]$.

based on some primitives operations into generation functions *without loosing completeness.*

4. Let us notice that when some cardinality results are provided, we are able to parameterize our generation functions in order to get uniform distributions for some non-trivial domains like some recursive structures.

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 ([29]) which is both a symbolic system and a programming language, in which our prototype is implemented.

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

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

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 ([7]) 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}]
```

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.

Cartesian Product
Given any tuple of simulation pairs (D_i, γ_i), it is easy to build the simulation pair $(\prod_i D_i, \gamma)$ where γ is the tuple whose i-th component is equal to γ_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 γ_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.

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 γ_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.

Mapping
Given any simulation pair (D, γ) 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.

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.

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, γ, the rejection method ([7]), 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 γ, 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 $\frac{\log(n)}{n}$.

Even if we do not provide any intersection operation as such, it is often possible to express the intersection of two sets as the subset of one of them and the above rejection method applies.

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 $\frac{1}{(MaxInt+1)(MaxInt-x+1)}$. We can also describe the set $\left\{(x,y) \in nat^2 |\ x \leq y\right\}$ by:

```
DNat := Sequence[{nat,
          Function[x,intInterval[x,INFINITY]]}]
```

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.

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 [3] for a more general presentation).

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 Σ:

```
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 Σ-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[3] 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

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

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}]
  ```

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:

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

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:

$$\begin{cases} head(x_0) = e_0 \\ size(x_0) \leq n \end{cases}$$

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:

$$\begin{cases} e_1 \leq head(x_1) \\ e_0 = e_1 \end{cases}$$

Let us consider the subset $G_{u,n}$ of $nat \times nat$ defined by:

$$\begin{cases} v_1 \leq v_2 \\ u = v_1 \end{cases}$$

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:

$$\begin{cases} head(x_1) = e_2 \\ size(x_1) \leq n-1 \end{cases}$$

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 $G_{u,n}$;
- on the range of the recursive function for which the equation is solved (*head* here).

AN EXAMPLE

We now will illustrate on an example of a List specification how to implement our approach of probabilistic functional testing. This specification will be written in the system OBJ which is a specification environment based on order sorted equational logic. A main advantage of OBJ is to provide a strong type system by using the notion of *subsort* which allows to support overloading, coercion, error handling. It facilitates the description of the domain of each formula.

The List Specification

We give below a List specification module (called a theory in OBJ) which specifies the abstract data type List by giving on the one hand the module interface (i.e. the set of the specified sorts and operations) and on the other hand properties on this module interface (i.e. well typed equations or conditional equations).

```
th List is sorts List EmptyList
                 NeList SortedList .
```

```
 subsorts SortedList < NeList <  List .
 subsorts EmptyList  < List .
 protecting NAT BOOL .
 op nil     : -> EmptyList .
 op cons__  : Nat List -> NeList .
 op cons__  : Nat EmptyList -> SortedList .
 op head_   : NeList -> Nat .
 op tail_   : NeList -> List .
 op ins__   : Nat SortedList -> SortedList .
 op sort_   : NeList -> SortedList
 op sorted_ : List -> Bool .
 var I J : Nat .  var N : NeList .
 var S : sortedList .  var L : List .
 eq head(cons(I,L)) = I .
 eq tail(cons(I,L)) = L .
 cq ins(I,cons(J,S)) = cons(I,cons(J,S))
          if sorted(cons(J,S)) and I <= J .
 cq ins(I,cons(J,S)) = cons(J,ins(I,S))
          if sorted(cons(J,S)) and J < I .
 eq sort(cons(I,nil)) = cons(I,nil) .
 eq sort(cons(I,N)) = ins(I,sort(N)) .
 eq sorted(nil) = false .
 eq sorted(cons(I,nil)) = true .
 eq sorted(cons(I,cons(J,L))) =
             (I <= J) and sorted(cons(J,L)) .
endth
```

We detail below some elements of the specification:

- `List`, `Emptylist`, `NeList` and `SortedList` are the new specified sorts introduced by the List module.
- `s1 < s2` declares that the sort `s1` is a subsort of the sort `s2`. It means that any term of sort `s1` may be also interpreted as a term of sort `s2`. For example, the non-empty lists of sort `NeList` and non-empty sorted lists of sort `SortedList` are also of sort `List`.
- The List module is defined by importing and protecting the NAT and BOOL modules of natural numbers and booleans where the operations `<=`, `<` and `and` are specified.
- Operations are declared, their type being defined by a Cartesian product of (sub)sorts for their domain and by a sort for their codomain. For example, the operations `tail` and `head` are only defined on non-empty lists. It prevents from having to manage exceptional terms such as `head(nil`.
- Axioms of the List data type express the required properties. They are introduced either by the notation `eq` or the notation `cq` depending on whether they are equations or conditional equations. The role of the `if` statement is to restrict the domain on which the principal equation holds. All these axioms are implicitly universally quantified with respect to all the variables occuring in them.
- An operation f can be specified by partitioning its domain into smaller subdomains by using pattern-matching on its arguments. Using patterns allows to give a syntactic description of some well chosen terms. For example, the operation `sorted` is first specified for the empty list (the pattern `nil`) , then for the list of length 1 (the pattern `cons(I,nil)`), and finally for the list of length strictly greater than 1 (the pattern `cons(I,cons(J,L))`). These three patterns cover all the cases of building a List term.

The axioms specify the operations of the List specification accordingly to the intended meaning. For example, `sorted` is a predicate checking whether a list is sorted or not.

Testing from a Specification

To simplify, a specification in OBJ is simply a set of axioms over a signature, which are formulas of the form (the `if` statement part being optional) :

$$f(exp_1, \cdots, exp_n) = exp \texttt{ if } cond$$

where the expressions exp_1, ... exp_n, exp and $cond$ depend on the variables x_1, ... x_n of the formula.

In order to consider such formulas within our setting of probabilistic functional testing, it remains to make clear on which domains such formulas hold.

Each variable occuring in the formula is declared in a given sort and, thus can be replaced by any term of this sort. As the domain covered by a variable corresponds to a term set defined by simple combination of enumeration set, Cartesian product or recursive structure, a sort defines a term set belonging to $\mathcal{S}$ (the set of the simulation pairs handled by our tool defined in the previous sections). For example, the variable `L` of sort `List` covers a free linear recursive structure defined by the the free constructor `cons`. In the previous section, we gave a simulation pair (Set_s, γ_s) for each sort s occuring in the List specification (i.e., for `Bool`, `Nat`, `EmptyList`, `NeList`, `SortedList` and `List`.)

Each pattern exp_i may be seen as a mapping from the term sets denoted by the formula variables on to a subset of Set_s where $s_1 \cdots s_n \rightarrow s$ is the type of f. For example, the pattern `cons(I,cons(J,L))` in the last axiom specifying `sorted` may be seen as a mapping from $Set^2_{\texttt{Nat}} \times Set_{\texttt{List}}$ to a subset of $Set_{\texttt{List}}$. Thus, from the previous section, there is no difficulty to build a generation function on the codomain of the patterns exp_i.

When the formula includes an `if` statement, there are several ways of defining the domain of the formula :

- The first one consists in considering that the formula is a composed one including the `if` statement and that its domain is simply described by the Cartesian product of the patterns exp_i codomain.
- The second one consists in considering that the formula to be considered is only an equation (i.e. $f(exp_1, \cdots, expr_n) = exp$) but that its domain is the subdomain of the whole previous Cartesian

product defined by the predicate *cond*. This subdomain is called the validity domain of the axiom.

For example, the first axiom specifying `ins` can be seen either as the formula:

$$i \leq j \Rightarrow ins(i, cons(j, s)) = cons(i, cons(j, s))$$

on the domain $Set^2_{\texttt{Nat}} \times Set_{\texttt{SortedList}}$ or as the formula:

$$ins(i, cons(j, s)) = cons(i, cons(j, s))$$

on the domain $DNat \times Set_{\texttt{SortedList}}$ where $DNat$ denotes the set $\{(x, y) \in Set^2_{\texttt{Nat}} | x \leq y\}$ (we gave a generation function for such a sequence set).

In the first case, the generation function is directly derived from the generation functions γ_s associated to each sort s. In the second case, if the predicate *cond* defines a subdomain D belonging to the class $\mathcal{S}$ of simulation pairs under the form (D, γ_D), then it suffices to use γ_D. Otherwise, one can apply the rejection method.

In any case, the axioms can be provided with a domain in such a way that the resulting formulas fit in with our setting of probabilistic functional testing. In order to test a system against a specification, one can test the system against each axiom and get a distinct (μ, ϵ, α)-validation for each axiom, depending on the relative importance given by the tester to each axiom. If the tester wants to provide a global (μ, ϵ, α)-validation, then it suffices to give weights to the axioms and then to draw test cases accordingly to these weights.

As a simple example and provided that the two parameters α and ϵ are given, if one wants to test the first axiom `head(cons(I,L)) = I`, it suffices to draw at least $\frac{\log(\alpha)}{\log(1-\epsilon)}$ couples (i, l) by using the generation function $(\gamma_{Nat}, \gamma_{List})$. The building of these generation functions are assisted by our tool as already explained in the previous sections.

Remark Let us recall that all this process is only possible under the hypothesis that the specified operations (resp. the equality predicates) can be submitted (resp. interpreted) by the system under test. But it may happen that it exports an equality predicate only for some sorts called *observable* (e.g. booleans or integers). For example, most of the time, when lists are implemented using pointers, the equality of two lists is only available if a specific equality predicate has been implemented. Nevertheless, lists can be observed through the observable contexts (for example the terms $head(tail^n(l))$). The set $\mathcal{C}$ of all the observable contexts allows to distinguish lists: in other words, two lists l_1 and l_2 are equal if and only if for each context[4] c of $\mathcal{C}$, $c[l_1] = c[l_2]$ (see [19] for more details on the impact of observability on testing). Finally, a formula such as:

$$\forall (i, j, s) \in D, sort(cons(i, n)) = ins(i, sort(n))$$

is replaced by the formula:

$$\forall (i, j, s, c) \in D \times \mathcal{C}, c[sort(cons(i, n)] = c[ins(i, sort(n))]$$

Now the set $\mathcal{C}$ of observable contexts is only an inductively defined set, therefore one can provide a function generation on $\mathcal{C}$. (see [3] for more explanations, in particular when $\mathcal{C}$ is not linear).

[4] $c[l_1]$ denotes the term c where l has been replaced by l_1.

RELATIONSHIP WITH SOME OTHER APPROACHES

Partition Testing

Partition testing ([15, 28, 26, 4]) 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 (μ, ϵ, α)-validation. More precisely, let us denote:

$$w_i = \frac{n_i}{\sum_j n_j} \qquad \bar{\mu} = \sum_i w_i \mu_i \qquad \bar{n} = \sum_i n_i$$

where μ_i is the uniform distribution on D_i. Then, we have the following result:

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

Proof: Let $\alpha \in]0, 1[$. For each subdomain D_i, the test of length n_i is a $(\mu_i, 1 - \sqrt[n_i]{\alpha}, \alpha)$-validation. This means that if one denotes by F_i the set $D_i \cap F$, the following holds:

$$\mu_i(F_i) \leq 1 - \sqrt[n_i]{\alpha}$$

where one has a probability of at most α to err. By linear combination of the previous inequalities with coefficient w_i, it follows that:

$$\bar{\mu}(F) \leq \sum_i w_i(1 - \sqrt[n_i]{\alpha})$$

where one errs if at least, one of the assertions about F_i is wrong. But one is then in the complementary of a set whose probability is at least $(1-\alpha)^n$. ■

Statistical Structural Testing

Statistical structural testing [25, 21] 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 μ 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 μ, partition D_i and test quality q, it is possible to deduce, for any given α 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.

Operational profile

The operational profile is a distribution, often invoked in software engineering, which models the actual client's relative use of the data in the input domain. When the client furnishes this distribution, say μ_{op}, then if μ_{op} is complete, it can of course be used to build a (μ, ϵ, α)-validation. However μ_{op} is not necessarily complete and it is suitable to also consider other distributions to check exceptional cases, such as ringing alarms, etc. To summarize, we think that the final distribution μ should be defined as a barycentre of several dedicated distributions. The choice of the relative weights of each distribution to get the final μ has to be negotiated between the vendor and the client.

Deterministic Functional Testing

Once a successful test has been conducted, if we want to increase the reliability evaluation in order to reach a greater confidence in the system under test, then 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 ("nominal" or "to the limits") 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 (μ, ϵ, α).

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 [20]. 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 $\mathcal{S}$ 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 primitive (as in our approach).

CONCLUSION AND PERSPECTIVES

We have defined a framework for *probabilistic functional testing*. Our first contribution is the formalization of the testing activity in term of (μ, ϵ, α)-validation. It allows to associate to any successful test of length N drawn according to the distribution μ two useful quantitative measures: ϵ which gives a probabilistic upper bound of the potential error domain and α 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. We also explain how one can generate appropriate distributions for data domains including intervals of integers, unions, cartesian products, inductively defined sets which are the most common domains in computer science. According to this theory, we have proposed a tool to assist test generation on theses domains. Our tool is only a first prototype which proves the applicability of our method. The main remaining difficulty is the ability to properly describe an input domain in term of the primitives offered by our tool. There are some ongoing researches in order to assist the activity of producing such domain descriptions, from the specification directly, and in such a way that the corresponding generated distributions are valuable from the testing point of view. Moreover, 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] A. Arnould, P. Le Gall, B. Marre: *Dynamic testing from bounded data type specifications.* Proc. of EDCC-2, Second European ependdable Computing Conference, Taormina, Italy. 1996.

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

[3] 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.

[4] 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.

[5] P.D. Coward: *A review of software testing.* Information and Software technology, U.K., Vol.30, No.3, Butterworth & Co Pub. Ltd, p.189-198. 1988.

[6] D. Daccunha-Castelle, M. Duflo: *Probabilités et Statistiques: Problèmes à temps fixe.* Masson. 1982.

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

[8] 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.

[9] R.K. Dong, Ph.G. Frankl: *The ASTOOT approach to testing object-oriented programs.* ACM Transactions on Software Engineering and Methodology, Vol.3, p.39. 1994.

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

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

[12] M.C. Gaudel: *Testing can be formal, too.* Proc. of TAPSOFT'95, Theory and Practice of Software Developement, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, Springer-Verlag LNCS 915. 1995.

[13] J.V. Guttag, J.J. Horning: *Report on the LARCH shared language.* Science of Computer Programming Journal, Vol.6, No.2, p.103-134. 1986.

[14] 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.

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

[16] T. Higashino, G.V. Bochmann: *Automated analysis and test case derivation for a restricted class of LOTOS expressions with data parameters.* IEEE Trans. on Software Engineering, Vol.9, p.29-42, January. 1994.

[17] C.B. Jones: *Systematic software development using VDM.* Prentice Hall. 1986.

[18] C. Kirchner, H. Kirchner, J. Meseguer: *Operational semantics of OBJ3.* Proc. of the 15th International Colloquium on Automata, Languages and Programming (ICALP), Springer-Verlag LNCS 317, p.287-301. 1988.

[19] P. Le Gall, A. Arnould: *Formal Specifications and Test: Correctness and Oracle.* Recent Trends in Data Type Specification, M. Haveraaen, O. Owe, O-J. Dahl eds, Springer-Verlag LNCS 1130, p.342-358. 1996.

[20] 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.

[21] 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.

[22] J.D. Musa, A. Iannino, K. Okumoto: *Software reliability : measurement, prediction, application.* Mc Graw-Hill, New-York. 1987.

[23] J.M. Spivey: *The Z notation: a reference manual.* Prentice Hall. 1989.

[24] P. Stocks, D.A. Carrington: *Test templates: A specification-based testing framework.* Proc. of the 15th Intl Conf. on Software Engineering, p.405-414, May. 1993.

[25] 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.

[26] 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.

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

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

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