An ongoing work on statistical structural testing via probabilistic concurrent constraint programming

Matthieu Petit* Arnaud Gotlieb
IRISA / INRIA
Campus Beaulieu
35042 Rennes cedex, FRANCE
{Matthieu.Petit,Arnaud.Gotlieb}@irisa.fr

Abstract

The use of a model to describe and test the expected behavior of a program is a well-proved software testing technique. Statistical structural testing aims at building a model from which an input probability distribution can be derived that maximizes the coverage of some structural criteria by a random test data generator. Our approach consists in converting statistical structural testing into a Probabilistic Concurrent Constraint Programming (PCCP) problem in order 1) to exploit the high declarativity of the probabilistic choice operators of this paradigm and 2) to benefit from its automated constraint solving capacity. This paper reports on an ongoing work to implement PCCP and exploit it to solve instances of statistical structural testing problems. Application to testing Java Card applets is discussed.

1 Introduction

Random testing. Random testing is an approach of software testing that is recognized to be of particular interest to improve software reliability [5]. Random testing has the great advantages 1) to introduce the randomness in the way the test data are selected and 2) to provide powerful statistical tools to evaluate the quality of the generated test set. The price to pay for these benefits concerns the necessity to have an automated oracle function, that is to say the availability of a procedure that can decide the correctness of the computed output. The most basic way of doing random testing consists in using a uniform probability distribution over the input domain of the program under test (every input point has the same probability to be chosen). In certain cases, this approach has even been demonstrated to be as competitive as other more sophisticated testing approaches [12, 6]. However, this is a somehow "blind" way of testing programs. Roughly speaking, it is clear that certain elements of the program have a very low probability to be activated and then they may not be exercised by using a uniform random generator as the number of test data to be generated has to remain reasonable.

*This work is part of the GENETTA project granted by the Brittany region

```
int foo(int x, int y) {
1.  if (x =< 100  &&  y =< 100)
    {
2.      if (y > x + 50)
3.          ...
4.      if (x * y < 100)
5.          ...
    }
```

Figure 1. Program foo

Motivating example. Consider the Java-like program of Fig.1 where x, y are restricted to take their values into $[0, 1000] \times [0, 1000]$ and suppose that we want to evaluate the interest of uniform random testing by looking at the coverage of the *all_paths* criterion. It is not difficult to see that path 1-2-3-4-5 of this program has a very low probability (near to one over ten thousands) to be exercised, as only 98 input points over 1001^2 satisfy all three conditions.

Statistical structural testing. To address this problem, Thévenod-Fosse and Waeselynck introduced the statistical structural testing technique [24] which aims at building a non-uniform input probability distribution to maximize the coverage of some structural criteria. In the above example, the problem consists in building a non-uniform distribution such as the theoretical probability to activate path 1-2-3-4-5 is equal to 20 percents as there are 5 execution paths. Although not strictly viewed as model-based testing, this

process can be formulated as building a model and generating test data from it by solving constraint systems [9]. The model is the control flow graph of the program under test where the edges are labelled with their expected theoretical probability to be activated. Algorithms to generate test data consist then to explore this model and extract probabilistic constraints over the input domain.

Using probabilistic concurrent constraint programming. Our approach targets to realize statistical structural testing by using Probabilistic Concurrent Constraint Programming (PCCP) [19, 11]. This paradigm extends Concurrent Constraint Programming [21] with probabilistic choice operators in order to implement randomized algorithms [19] or stochastic processes [10]. We want 1) to exploit the high declarativity of these probabilistic choice operators and 2) to benefit from the automated constraint solving capacity of this paradigm to address the problem mentioned above. This paper reports on our implementation of PCCP over finite domains constraints and on a first proposition to interpret statistical structural testing as a PCCP problem.

Application to testing java card applets. Our application domain is the smart card quality assurance field. Research works on software testing in that field includes the BZ-testing approach designed by Legeard and Peureux [14, 1] to generate automatically test cases from a formal B or Z specification of the Java Card transaction mechanism. In [20, 18], Pretschner et al. make use of constraint solving techniques to generate test cases for validating the authentification protocol of an inhouse smart card. At the same time, Clarke et al. [3] developed symbolic test generation algorithms to generate on-the-fly test cases for a feature of the CEPS[1] e-purse application and Martin and Du Bousquet [15] proposed to use UML-based tools to generate test suites for testing Java Card applets. All these approaches have in common to require first a formal model (B specification, automata, input/output transition system or statecharts) to be constructed in order to generate test cases. Sometimes this effort is too costly and cheaper approaches such as random testing and statistical structural testing are preferable. For testing java card applets, our approach only asks for a probabilistic control flow graph to be extracted from a code applet analysis. Hence, our approach appears as being complementary with other approaches. Note that generating test data from a formal model does not guarantee the code structure to be completely covered, which is the ideal goal of statistical structural testing.

Contents. The paper is organized as follows : Section 2 describes statistical structural testing as model-based testing ; Section 3 introduces our implementation of PCCP over finite domains constraints whereas Section 4 contains a first proposition to interpret statistical structural testing as a PCCP problem. Section 5 introduces potential applications to testing Java Card applets and finally Section 6 discusses the remaining problems to fulfill the gap between theory and practice.

[1]The Common Electronic Purse Specification is a standard for creating inter-operable multi-currency smart card e-purse systems.

2 Statistical Structural Testing as Model-based Testing

2.1 Background

Given a structural criterion C, S_C is the set of elements which must be covered during testing. By using a random test data generator, the probability to exercise element $k \in S_C$ is noted p_k. By a simple probabilistic reasoning, the statistical testing efficiency is characterized with the number N of generated test data and p_{min} the lowest probability of an S_C element to be activated. This efficiency parameter is usually referred to as the test quality and is noted q_N [24]. C is said to be covered with a probability q_N iff each element of S_C has a probability at least q_N to be activated during a N test data random generation.

The following relation holds [24] :

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

The relation can easily be justified since $(1 - p_{min})$ is the probability of the event : "the less probable element is not activated" then $1 - q_N$ is the probability of no less probable element activation during N test data random generation. As an immediate corollary, we get an estimation of the number of test data generation N_{min} to reach a selected value of the test quality q_N :

$$N_{min} = \left\lfloor \frac{\ln(1 - q_N)}{\ln(1 - p_{min})} \right\rfloor + 1$$

where $\lfloor x \rfloor$ denotes the integral part.

Statistical structural testing aims at finding an input distribution which maximizes p_{min} according to a given structural criterion C and a test quality q_N.

2.2 Probabilistic control flow graph

Although not strictly viewed as model-based testing, statistical structural testing can be formulated as such by the following two steps :

1. Building a model from a source code analysis and the computation of theoretical probabilities;

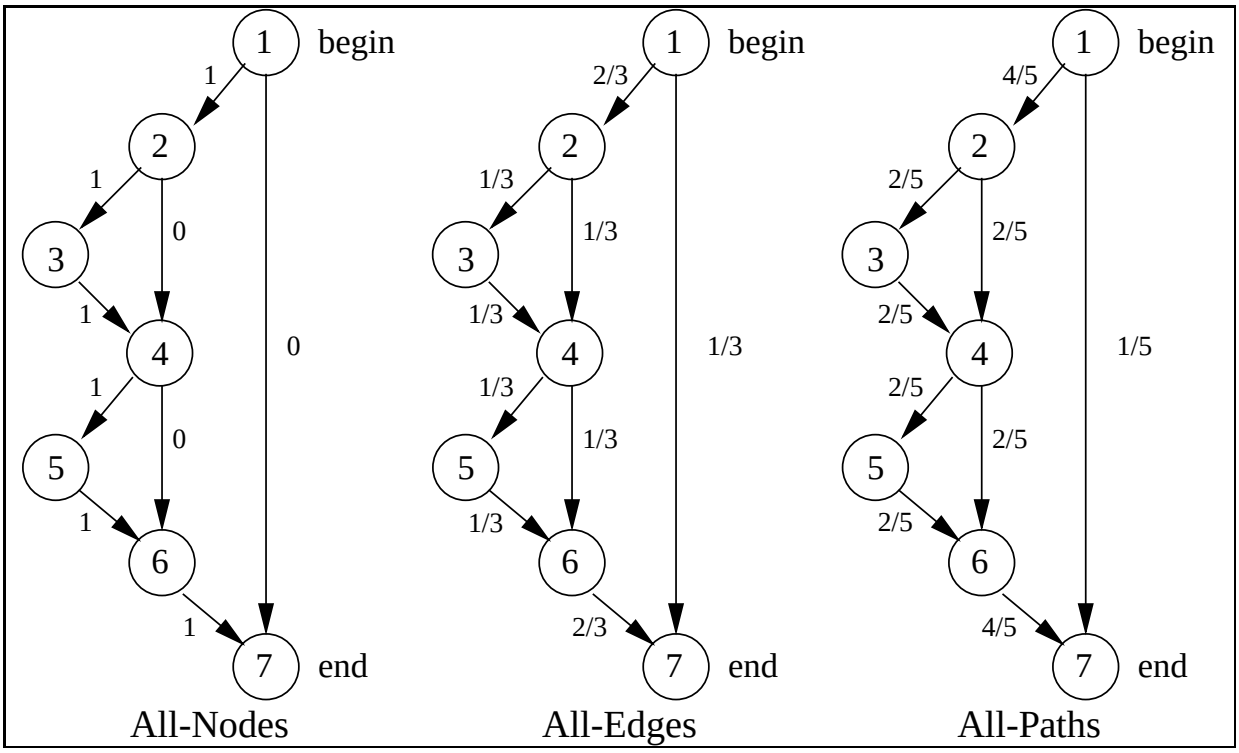

Figure 2. Probabilistic CFG of the foo program for All-Nodes, All-Edges and All-Paths criteria

2. Deriving an input probabilistic distribution from model runs and probabilistic constraint solving.

The model is based on an extended version of the control flow graph that can be extracted from any imperative and sequential program. However several restrictions must be taken into account. The integer types are treated but the strings and the float types are not. Aliasing, dynamic data structures, methods calls, exceptions, ... are not considered in the rest of the paper.

Definition 1. A control flow graph (CFG) is a connected oriented graph $\langle N, E, begin, end \rangle$ where each node represents a statement block and each edge represents a control transfer. N is the nodes set, E is the edges set, $begin \in N$ is the input node and $end \in N$ is the output node[2].

In this paper, only three classical CFG-based coverage criteria are considered : All-Nodes, All-Edges and All-Paths [25].

Definition 2 (All-Nodes criterion). A set Π of execution paths satisfies the All-Nodes criterion iff for each node $n \in N$, there is at least one path p in Π such that node n is on the path p.

[2]It is assumed that programs have only a single return point.

Definition 3 (All-Edges criterion). A set Π of execution paths satisfies the All-Edges criterion iff for each edge $e \in E$, there is at least one path p in Π such that p contains the edge e.

Definition 4 (All-Paths criterion). A set Π of execution paths satisfies the All-Paths criterion iff Π contains all execution paths from the $begin$ node to the end node in the CFG.

The CFG is extended with edge labels to build a model for statistical structural testing. A label represents the probability of an edge e to be activated, that is to say the theoretical probability of an input point which activates e to be chosen.

Let us recall that when C is a structural criterion, S_C denotes the set of elements which must be covered during the test and p_k denotes the probability to activate $k \in S_C$. A probabilistic control flow graph is a control flow graph $\langle N, E, begin, end \rangle$ extended by $L_C = \{p(e) \,|\, e \in E\}$ where $p_{min} = \text{Min}_{k \in S_c}(p_k)$ is maximized.

This probabilistic CFG can be considered as the test purpose for statistical structural testing. The goal is to construct an input probability distribution which attempts to reach this purpose where launching the random test data generation.

Fig. 2 contains three probabilistic CFG associated to the foo program. Note that p_{min} can easily be computed from these probabilistic CFGs :

Criterion	$\mathbf{p_{min}}$	**Less probable elements to activate**
All-Nodes	1	1, 2, 3, 4, 5, 6, 7
All-Edges	$\frac{1}{3}$	2-3, 2-4, 3-4, 4-5, 4-6, 5-6, 1-7
All-Paths	$\frac{1}{5}$	1-2-3-4-5-6-7, 1-2-3-4-6-7, 1-2-4-5-6-7, 1-2-4-6-7, 1-7

For all the three criteria, the same algorithm can be employed to compute the set of theoretical probabilities L_C . This algorithm does not present any difficulty and has been extensively described in [17]. In the presence of loops and non-feasible paths[3], the theoretical probabilities are difficult to reach by a random test data generation. In fact, in our current approach the theoretical probabilities are only computed by using static informations (such as the CFG). A probable improvement of our approach would be to consider theoretical probabilities that take into account dynamic informations.

Based on the probabilistic CFG, the input probability distribution must be constructed. In the following, we explain how this problem can be transformed into a PCCP problem.

3 A new Probabilistic Concurrent Constraint Programming implementation

3.1 Concurrent Constraint Programming

We start by recalling the main principles of the concurrent constraint programming paradigm [21, 22].

Processes are the main notion in concurrent programming. Processes are programs that are executed concurrently and that can interact with each other. In concurrent constraint programming, concurrent processes communicate via a common constraint store. The constraint store is a conjunction of constraints on the possible values of variables.

The processes are constructed with the three following operators : **tell**(C), **if** C **then** $Process$ and $Process \| Process$. **tell**(C) imposes the constraint C to the constraint store. **if** C **then** $Process$ imposes the constraints of $Process$ if C is entailed by the constraint store. $\|$ is the parallel composition operator.

Concrete implementations of this paradigm include cc(FD) [13], Oz [23] or clp(FD) [2] just to name a few. We have chosen to build our framework upon clp(FD). Hence, in the following a $Process$ will be viewed as a $Goal$, **if then** as an **ask** operator, $\|$ as the conjunction of clp(FD). Note that variables are constrained to take their values from given finite sets (for example integer values) in clp(FD).

[3]path of the CFG that cannot be executed by any test datum

3.2 Probabilistic choice operators

Gupta et al. pioneered the inclusion of a probabilistic choice operator in concurrent constraint programming [11]. The probabilistic choice is introduced as a local random variable into a process [10]. Based on this operator, we present its implementation in clp(FD).

Syntax.

```
choose(X,[V1,..,Vn],[W1,..,Wn],Goal)
```

where `X` is a local random variable of `Goal`, `[V1,..,Vn]` represents a finite set of possible values and `[W1,..,Wn]` represents a probability distribution. `V1,.., Vn, W1,.., Wn` are finite domain variables.

`choose(X,[V1,..,Vn],[W1,..,Wn],Goal)` is true iff `X=Vi,Goal` is true with the probability p_i where $p_i = \frac{\text{Wi}}{\sum_{j=1}^{n} \text{Wj}}$.

Let us give an example of the usage of this operator :

Example 1 (extracted from [10]).

```
?-choose(X,[0,1],[1,1],[X#=Z]),
  choose(Y,[0,1],[1,1],
           [ask(Z#=1,[Y#=1])]).
```

After the example running, Z is constrained to 0 with a probability $\frac{1}{2}$ *(event* $X = 0$*), to 1 with a probability* $\frac{1}{4}$ *(event* $X = 1 \wedge Y = 1$*) and the goal fails with a probability* $\frac{1}{4}$ *(event* $X = 1 \wedge Y = 0$*).*

Here is the result of `prob_config_term/3` *which computes the probability to reach the different constraint stores after 5000 goal executions.*

```
R=[([Z=0],0.5084),
   (([Z=1],0.2434),([no],0.2482)]
```

Each element of `R` *represents the domain of* Z *after the goal execution and the probability to reach this constraint store. The domain* `[no]` *represents the Prolog's standart fail which corresponds to an inconsistent constraint store.*

The originality of our implementation comes from the fact that probabilistic choice operators are defined as global constraints in clp(FD). Global constraints are a good way for giving global semantics to complex constraints. Furthermore, such operators appear to the user like single constraints and so can be awaked and treated efficiently by the constraint propagation algorithm.

Note that this probabilistic choice operator introduced by Gupta et al. did not allow the domain and the distribution to be unknown. Our work also consisted in extending this view, as explained in section 3.4.

3.3 Constraint solving

In clp(FD), constraint solving is based on constraint propagation and labelling.

The constraint propagation permits to prune the domain of the variables. This reduction is usually done with interval reasoning. When a constraint can not be used to prune the current domains of its variables, it falls into an "asleep" state. The constraint is said "awaken" when another constraint reduces the domain of one of its variables. Then, this new information is used to reduce the variable domain of the constraint.

After constraint propagation, an enumeration is used to get a complete solution of the constraint system. This is done by a labelling process which tries to give a value to a variable one by one and propagates throughout the constraint system. This process is repeated until all the uninstantiated variables become instantiated. If this valuation leads to an inconsistency then the process tries an other possible value.

3.4 Extensions of the probabilistic operators

In regards with the statistical structural testing problem, the input probability distribution is the unknown parameter of the problem. In this context, we decided to extend the probability choice operators family based on the operator `choose/4` with unbound domain and unbound distribution. When the domain variables (`[V1,..Vn]`) or the probability distribution variables (`[W1,..Wn]`) are unbound, the probabilistic choice operators as global constraints fall into the "asleep" state, because the probabilistic choice is not feasible. Two new global constraints of the same family have been introduced : `choose_unif/4` and `choose_dec/5`.

Syntax.

```
choose_unif(X,Xmin,Xmax,Goal)
```

where `X` is a random variable, `Xmin` and `Xmax` are finite domain variables

`choose_unif(X,Xmin,Xmax,Goal)` is true iff `X=Val,Goal` is true with the probability $\frac{1}{n}$ where $n =$`Xmax-Xmin`$+1$ and `Val in Xmin..Xmax`.

The global constraint `choose_unif/4` is fallen into an "asleep" state when `Xmin` or `Xmax` are uninstantiated. In this case, we exploit partial informations on the `X` domain by telling the constraints : `Xmin#=<X`, `X#=<Xmax` and `Xmin#=<Xmax`. The algorithm used to compute `choose_unif/4` is based on a fix point computation that tries iteratively to prune the domain of `Xmin` and `Xmax`.

Let us illustrate the use of unbound variables in `choose_unif/4` with the following example.

Example 2.

```
| ?- choose_unif(X,0,Xmax,[Xmax#=X-1]).
```

`Xmax` *is unknown then the random generation of a value for* `X` *is not feasible. The constraint falls into the "asleep" state. However, the goal*

```
Xmax in 0..1000
```

awakes the constraint and the domain reduction of `Xmax` *during the fix point computation leads to*

iteration number	`Xmax` *domain*
1	*0..999*
2	*0..998*
⋮	⋮
999	*0..1*
1000	*0..0*
1001	***no***

The 1001^{ith} *iteration of the fix point computation gives an inconsistency of the constraint without any probabilistic choice. So, the* `Xmax` *domain becomes empty.*

Finally, the new global constraint `choose_dec/5` is introduced to simulate the conditional statement behavior in clp(FD).

Syntax.

```
choose_dec(C,W1,W2,Goal1,Goal2)
```

where `W1` and `W2` are finite domain variables.

`choose_dec(C,W1,W2,Goal1,Goal2)` is true iff `C,Goal1` is true with the probability p or $\neg$ `C,Goal2` is true with the probability $1-p$ where $p = \frac{W1}{W1+W2}$.
The global constraint is "asleep" when `W1` and `W2` are not valuated. For this operator, the probability distribution is not necessarily bound.

4 Statistical structural testing as a PCCP problem

To address the problem of deriving an input probability distribution from model runs, we translate a program into a clp(FD) goal. A random solution of this goal can be seen as a random test datum generation for the statistical structural testing. We follow the approach of [8] where the SSA form is used.

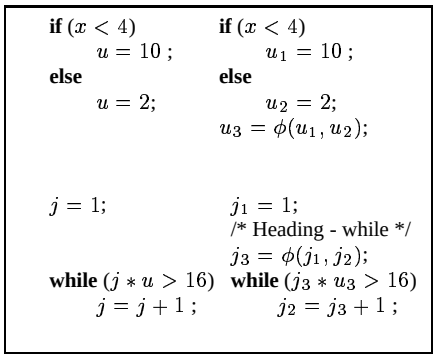

Figure 3. SSA form of control statements

4.1 Static Single Assignment form

The SSA form is a version of a procedure on which every variable has a unique definition and every use of a variable is reached by this definition [4]. The SSA form of a basic block is obtained by a simple renaming ($i = i + 1$ leads to $i_2 = i_1 + 1$). For the control structures, SSA form introduces special assignments, called ϕ-functions, to merge several definitions of the same variable. For example, the SSA form of the `if_then_else` is illustrated in the top of Fig. 3. The ϕ-function of the statement $u_3 = \phi(u_1, u_2)$ returns one of its argument : if the flow comes from the *then*-part then the ϕ-function returns u_1, otherwise u_2.
For other structures such as loop, the ϕ-functions are introduced in a special heading, as exemplified in Fig. 3. If the flow comes from the statement j_1 =..., then the ϕ-function returns j_1. On the contrary, if the flow comes from the body of the loop (j_2 =...) then, the ϕ-function returns j_2. The SSA form allows the statements to be reinterpreted as constraints with some restrictions for the while statement [8].

4.2 Translation

Assignment statement. The statement $x := expr$ is translated into `X#=E` where `E` is the syntactic translation of $expr$

Compound statement. The statement $Stmt_1; Stmt_2$ is translated into a conjunction (,) of two goals where `Goal1` (resp. `Goal2`) is the translation of $Stmt_1$ (resp. $Stmt_2$).

Conditional statement. The statement **if** b **then** $Stmt_1$ **else** $Stmt_2$ is translated into `choose_dec(C,W1,W2,Goal1,Goal2)` where `C` is the syntactic translation of b, `Goal1` (resp. `Goal2`) is the translation of $Stmt_1$ (resp. $Stmt_2$). `W1` and `W2` are computed from the theoretical probabilities of the probabilistic CFG.

Loop statement The statement **while** b **do** $Stmt$ is treated as a conditional statement with a recursive call. In our translation, the statement **while** b **do** $Stmt$ is considered as the statement **if** b **then** ($Stmt$;**while** b **do** $Stmt$). A lazy form of unfolding loop is used in our current framework but we plan to use a global combinator to simulate the while statement behavior .

4.3 Illustration of the translation

We illustrate the translation of the foo program. We suppose that `X` and `Y` are constrained to 0..1000 (input domain of foo program).

For All-Nodes criteria, the translation of the foo program is :

```
choose_dec(X#=<100#Y#=<100,1,0,
  [choose_dec(Y#>X+50,1,0,[],[]),
   choose_dec(Y*X#<100,1,0,[],[])],[])
```

We obtain as result of `prob_config_term/3`

```
R=[([X in 0..1,Y in 51..100],1.0)]
```

For All-Edges criteria, the translation of the foo program is :

```
choose_dec(X#=<100#/\Y#=<100,2,1,
  [choose_dec(Y#>X+50,1,1,[],[]),
   choose_dec(Y*X#<100,1,1,[],[])],[])
```

We obtain as result of `prob_config_term/3`

```
R=[([X in 0..1,   Y in 51..100],0.1610),
   ([X in 1..49,  Y in 52..100],0.1768),
   ([X in 0..100, Y in 0..100], 0.1640),
   ([X in 1..100, Y in 1..100], 0.1664),
   ([X in 0..1000,Y in 0..1000],0.3318)]
```

For All-Paths criteria, the translation of the foo program is :

```
choose_dec(X#=<100#/\Y#=<100,4,1,
  [choose_dec(Y#>X+50,1,1,[],[]),
   choose_dec(Y*X#<100,1,1,[],[])],[])
```

We obtain as result of `prob_config_term/3`

```
R=[([X in 0..1,   Y in 51..100],0.2126),
   ([X in 1..49,  Y in 52..100],0.1956),
   ([X in 0..100, Y in 0..100], 0.2004),
   ([X in 1..100, Y in 1..100], 0.1954),
   ([X in 0..1000,Y in 0..1000],0.1960)]
```

These results show that our translation respects the theoretical probabilities of the probabilistic CFG.

4.4 Random data test generation

We turn now to the design of the random test data generator itself.

After the translation of the imperative program into a PCCP(FD) goal, solving the resulting constraint system with a random labelling is equivalent to the random test data generation. A random labelling is a process that selects a tuple of values for input variables at random (predicate `random_lab\1`). Note that we are just interested by the first found solution of the constraint sytem.

The following example illustrates a possible random datum test generation for the foo program under the All-Edges criterion.

```
?-choose_dec(X#=<100#/\Y#=<100,4,1,
   [choose_dec(Y#>X+50,1,1,[],[]),
    choose_dec(Y*X#<100,1,1,[],[])],[]),
  random_lab((X,Y)).

 X=36,
 Y=1 ?
yes
```

5 Application to statistical structural testing of Java Card applets

In this section, we discuss the interest of testing Java Card applets with statistical structural testing via probabilistic concurrent constraint programming. As previously said, a lot of work has been carried out to automatically derive test cases from applet's formal specifications. Our approach distinguishes itself by making use of (non-uniform) random testing to address the code coverage problem without requiring a formal model to be constructed. This can be illustrated by the following method extracted from the well-known SUN Java Card applet `Wallet.java`.

```
private void credit(APDU apdu) {

// access authentication
 if( ! pin.isValidated() )
    ISOException.throwIt(
      SW_PIN_VERIFICATION_REQUIRED);
 byte[] buffer = apdu.getBuffer();

// Lc byte denotes the number of bytes in the
// data field of the command APDU
 byte numBytes = buffer[ISO7816.OFFSET_LC];

// indicate that this APDU has incoming data
// and receive data starting from the offset
// ISO7816.OFFSET_CDATA following the 5 bytes.
 byte byteRead =
       (byte)(apdu.setIncomingAndReceive());

// it is an error if the number of data bytes
// read does not match the number in Lc byte
 if(( numBytes != 1 ) || (byteRead != 1) )
  ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

// get the credit amount
 byte creditAmount =
  buffer[ISO7816.OFFSET_CDATA];

// check the credit amount
 if(( creditAmount > MAX_TRANSACTION_AMOUNT)
    || ( creditAmount < 0 ) )
  ISOException.
      throwIt(SW_INVALID_TRANSACTION_AMOUNT);

// check the new balance
 if((short)(balance+creditAmount)>MAX_BALANCE)
  ISOException.throwIt(SW_EXCEED_MAXIMUM_BALANCE);

// credit the amount
 balance = (short)(balance + creditAmount);
}
```

In Java Card programming, an input test datum is instantiated by giving values to the APDU[4] buffer. APDU is an ISO-normalized communication format between the card and the off-card applications. In its most general form, an APDU has the structure shown in Fig.4.

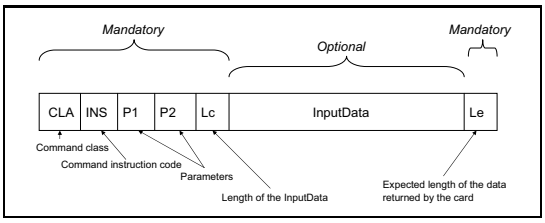

Figure 4. APDU command

Note that the code of the credit's method does not make too much assumptions on the form of the APDU command as it starts by checking several constraints on it. For example, it checks the expected size of the $InputData$ set by looking at the value of `byte numBytes = buffer[ISO7816.OFFSET_LC];`. If a uniform random generator is used, the event {`numBytes != 1`} will have a greater probability ($\frac{2^8-1}{2^8}$) than the opposite event {`numBytes == 1`} resulting in a very frequent execution of statement `ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);`. Statistical structural testing based on the translation shown in Sec.4 will handle this case by giving the same probability to both events.

We argue that PCCP over finite domain constraints is well adapted to the treatment of Java Card testing problems as most computations within an applet are done over integer types as strings or floats do not belong to the language. On the one hand, conditions such as

[4] Application Protocol Data Unit

`if(balance+creditAmount>MAX_BALANCE)` are efficiently handled by probabilistic concurrent constraint programming over finite domains as constraint propagation is well suited to solve arithmetic constraints. But on the other hand, it could be argued that statistical testing always requires an automatic way of checking the correctness of the computed results. For example, in the credit's applet example, how could we check if the returned balance is correct (when the flow reaches the last statement) without any formal specification ? This problem, known as the oracle problem, can be handled in at least two ways in java card programming. Firstly, some properties of the applet under test can serve as partial oracles. Examples of such properties include symmetry relations [7] or JML assertions [16], that are less demanding than full formal specifications. Secondly, older versions or prototype versions of the applet under test can serve as test oracles when they are available. This approach is usually referred to as regression testing.

6 Further work

This paper has introduced statistical structural testing via Probabilistic Concurrent Constraint Programming. This latter paradigm has been implemented and experienced over several examples although several extensions remain to be implemented. A first version of an automated translation of imperative programs to PCCP programs over finite domains has also been introduced and application to testing Java Card applets is foreseen. Nevertheless, the remaining difficulties reside in 1) the treatment of loops because the general iteration cannot be directly modelized as a global constraint due to the non termination problem ; 2) the treatment of polymorphic method calls as our translation is currently solely based on static informations and dynamic typing information is required ; 3) dealing with aliasing of references is mandatory but requires dynamic flow informations to be handled without approximation.

References

[1] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, M. Utting, and N. Vacelet. Bz-testing-tools: A tool-set for test generation from z and b using constraint logic programming. In *In Proc. of FATES'02, Formal Approaches to Testing of Software*, Brn, Czech Republic, Aug. 2002.

[2] M. Carlsson, G. Ottosson, and B. Carlson. An Open–Ended Finite Domain Constraint Solver. In *Prog. Languages, Implementation, Logics, and Programs (PLILP)*, Southampton, UK, Sep. 1997.

[3] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. Automated test and oracle generation for smart-card applications. In *In Int. Conf. on Research in Smart Cards (e-Smart'01)*, Springer Verlag, LNCS 2140, pages 58–70, 2001.

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. on Prog. Lang. and Sys.*, 13(4):451–490, 1991.

[5] Joe Duran and Simeon Ntafos. An evaluation of random testing. *IEEE Trans. on Soft. Eng.*, 10(4):438–444, Jul. 1984.

[6] P.G. Frankl, R.G. Hamlet, B. Littlewood, and L. Strigini. Evaluating testing methods by delivered reliability. *IEEE Trans. on Soft. Eng.*, 24(8):586–601, August 1998.

[7] A. Gotlieb. Exploiting symmetries to test programs. In *IEEE International Symposium on Software Reliability and Enginering (ISSRE)*, Denver, CO, USA, Nov. 2003.

[8] A. Gotlieb, B. Botella, and M. Rueher. Automatic Test Data Generation Using Constraint Solving Techniques. In *ACM Int. Symp. on Soft. Testing and Analysis (ISSTA). Soft. Eng. Notes*,23(2):53-62, 1998.

[9] S-D. Gouraud, A. Denise, M-C. Gaudel, and B. Marre. A new way of automating statistical testing. In *16th IEEE Int. Conf. on Automated Software Engineering (ASE'01)*, San Diego, CA, Nov. 2001.

[10] Vineet Gupta, Radha Jagadeesan, and Prakash Panangaden. Stochastic processes as concurrent constraint programs. In *ACM Princ. of Prog. Lang. (POPL)*, pages 189–202, New York, 1999. ACM.

[11] Vineet Gupta, Radha Jagadeesan, and Vijay A. Saraswat. Probabilistic concurrent constraint programming. In *International Conference on Concurrency Theory*, pages 243–257, 1997.

[12] Dick Hamlet and Ross Taylor. Partition testing does not inspire confidence. *IEEE Trans. on Soft. Eng.*, 16(12):1402–1411, December 1990.

[13] Pascal Van Hentenryck, Vijav Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(fd). Technical Report CS-93-02, Brown University, 1993.

[14] B. Legeard and F. Peureux. Generation of functional test sequences from b formal specification : presentation and industrial case study. In *Int. Conference on Automated Software Engineering (ASE'01)*, San Diego, USA, Nov. 2001.

[15] H. Martin and L.d. Bousquet. Automatic test generation for java-card applets. In Isabelle Attali and Thomas P. Jensen, editors, *Java on Smart Cards: Programming and Security, Works. JavaCard 2000, Cannes, France, 2000*, volume 2041 of *LNCS*, pages 121–136. Springer, 2001.

[16] H. Meijer and E. Poll. Towards a full formal specification of the java card api. In Isabelle Attali and Thomas P. Jensen, editors, *Smart Card Programming and Security, Int. Conf. on Research in Smart Cards*, volume 2140 of *LNCS*, pages 165–178. Springer, 2001.

[17] Matthieu Petit. Utilisation de la Programmation Concurrente par Contraintes Probabilistes pour le Test Statistique Structurel. Rapport de DEA d'Informatique, Université de Rennes 1, 2004.

[18] J. Philipps, A. Pretschner, O. Slotosch, E. Aiglstorfer, S. Kriebel, and K. Sholl. Model-based test case generation for smart cards. In *In Formal Methods for Industrial Critical Systems, Elec. Notes in Theoretical Computer Science 80*, pages 168–182, Trondheim, Jun. 2003.

[19] Alessandra Di Pierro and Herbert Wiklicky. An operational semantics for probabilistic concurrent constraint programming. In *International Conference on Computer Languages*, pages 174–183, 1998.

[20] A. Pretschner, O. Slotosch, H. Ltzbeyer, E. Aiglstorfer, and S. Kriebel. Model based testing for real: The inhouse card case study. In *Int. Works. on Formal Methods for Industrial Critical Systems (FMICS'01)*, pages 79–94, Paris, FR, Jul. 2001.

[21] Vijay A. Saraswat. *Concurrent Constraint Programming*. The MIT Press, 1993.

[22] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. Semantic foundations of concurrent constraint programming. In *ACM Princ. of Prog. Lang. (POPL)*, pages 333–352, Orlando, 1990. ACM.

[23] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

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

[25] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.