25

Test Selection for Object-Oriented Software Based on Formal Specifications

C. Péraire, S. Barbey and D. Buchs

Swiss Federal Institute of Technology, Software Engineering Laboratory, 1015 Lausanne, Switzerland, phone: +41 (21) 693.52.43, fax: +41 (21) 693.50.79, email: {Cecile.Peraire, Stephane.Barbey, Didier.Buchs} @ epfl.ch, url: hhtp:// lglwww.epfl.ch

Abstract

We propose a method of test selection based on formal specifications, for specification-based testing of object-oriented software. This method is based on rigorous theoretical foundations. To limit the size of test sets, we present several hypotheses specific to object-oriented systems, used to reduce an exhaustive test set into a pertinent test set. Regularity hypotheses are used to constrain the shape of test cases while uniformities, with or without subdomain decomposition, are used to limit the values of the variables appearing in the test cases. Care is taken to provide a method for which operational techniques can be implemented.

Keywords

Specification-based testing, test set selection, object-orientation, formal methods, operational methods.

Programming Concepts and Methods D. Gries, & W-P. de Roever (Eds.)

1 INTRODUCTION

Testing is obviously one of the answers to address the issue of quality in software, even developed using object-oriented (O-O) methods. However, given the complexity of today's software, a good test selection cannot be performed without a rigorous foundation for the theoretical, methodological and operational aspects.

For specification-based testing of O-O software, test selection is made difficult because the behavior of the objects does not only depend on the input domain, but also on the history of the objects, because of their persistent state. Thus, operations must not be tested individually, but in interaction. This combinatorial explosion implies to carefully determine that only necessary test cases are provided.

In this paper, we propose a testing method to take this issue into account. It is based on the theory of testing presented in [Barbey et al., 1996]. This theoretical framework, is an adaptation to O-O software of the Bernot, Gaudel and Marre theory of testing [Bernot et al., 1991]. Its essence is to reduce an exhaustive test set into a finite and pertinent test set by applying reduction hypotheses. The aim of this paper is mainly to show how to determine test shapes that are appropriate for the system under test, and to describe some of the applicable hypotheses for O-O software, while taking care that the test selection can be (semi-) automated.

In section 2, we present other research that addresses the same topic. (A comparison with our approach is made in the conclusion.) The third section is devoted to the presentation of already published results about our approach [Barbey et al., 1996] as well as a short informal introduction to our specification language, *CO-OPN/2*. The fourth section presents the test selection process, which is based on choosing hypotheses of two kinds — regularity and uniformity— and deriving constraints from those choices. Examples of hypotheses, together with their strategy, are described in section 5. In section 6, we present a refined uniformity hypothesis which increases the quality of the test sets by performing subdomain decomposition.

2 STATE OF THE ART

Many methods have already been proposed for the test selection of O-O software. A good survey of the state of the art can be found in [Binder, 1996]. For the purpose of this article, we will however limit ourselves to the two methods that are closer to ours.

The ASTOOT method [Doong and Frankl, 1994] proposes a testing strategy and a tool based on traces for specifications written in an algebraic O-O language, LOBAS. A test case consists of pairs of sequences of calls to the operations of one class, and a flag indicating the relationship between the sequences (equivalent or not). The test set generation is performed by selecting interesting sequences of operations and generating equivalent sequences by term rewriting of the axioms of the specification. Thus, the test sequences are directly derived from the specification. The test set generation is partly automated, but the equivalence relationship (oracle) must be provided by the tester.

Another method has been proposed by Kirani [Kirani, 1994]. This approach is based on the specification of MtSS, Method Sequence Specification, which

describes valid sequences of invocations of operations of a single class under the form of a regular expression. Those regular expressions can be developed from many kinds of specifications such as state transition diagrams or object diagrams. Given these expressions, test cases can be selected by generating all possible sequences of invocations. The test set is reduced by using standards methods, such as random, partition and data flow testing. However, this method only includes a strategy for generating sequences, and does not cover the choice of parameter values. Moreover, Kirani does not tackle the development of an oracle for his method.

3 THEORETICAL GROUNDS

Specification-based testing is an approach to find errors in a program by verifying its functionalities, without analyzing the details of its code, but by using the specification of the system only. It can be summarized as the equation:

$$(P \not\models_O T \Rightarrow P \not\models SP)$$

i.e. that the test set T applied on a program P will reveal that P does not implement correctly the specification SP. The goal in selecting T is to uncover the cases where the program does not satisfy the test cases, and thus reveal errors wrt the specification.

Test selection is based on the properties of the specification language, which must be theoretically well founded. Usually, specification languages have a notion of formula representing the properties that all desired implementations satisfy. Test cases can be expressed using the same language, however this is not necessary. The most interesting solution is to have a specification language well adapted to the expression of properties, and another language to describe test cases that can be easily applied to an oracle, as long as there is a full agreement between these two languages. In the remaining of this section, we will present a specification language, *CO-OPN/2*, and a language for expressing test cases, *HML*.

3.1 Specifying Object Systems with *CO-OPN/2*

This section presents a concurrent O-O specification language, called *CO-OPN/2* (Concurrent O-O Petri Nets) that will be used to demonstrate the testing principles. *CO-OPN/2* is a formalism developed for the specification and design of large O-O concurrent systems [Biberstein et al., 1997]. Such system consists of a possibly large number of entities, which communicate by triggering parameterized events (sending messages). The events to which an object can react are also called its methods. The behavior of the system is expressed with algebraic Petri nets.

A *CO-OPN/2* specification consists of a collection of two different kinds of modeling entities: algebraic abstract data types (*ADTs*) and classes. Algebraic sorts are defined together with related functions, in *ADT* modules. Algebraic sorts are used to specify values such as the primitive sorts (integer, boolean, enumeration types, etc.) or purely functional sorts (stacks, etc.). Class' type sorts are defined together with their methods in a distinct class module. Such a module corresponds to the notion of encapsulated entity that holds an internal state and provides the outside with various services.

Cooperation between objects is performed by means of a synchronization mechanism, i.e. each event may request synchronization with the methods of one or of a group of partners using a synchronization expression. Three synchronization operators are defined: "//" for simultaneity, ".." for sequence, and "+" for alternative. The syntax of the behavioral axiom that includes synchronization is

[Condition =>] Event [**with** SynchroExpression]: Precondition -> Postcondition

Condition is a condition on algebraic values, expressed with a conjunction of equalities between algebraic terms. Event is an internal transition name or a method with term parameters. SynchroExpression is the (optional) expression described above, in which the keyword **with** plays the role of an abstraction operator. Precondition and Postcondition correspond respectively to what is consumed and to what is produced in the different places within the net (*in* arcs and *out* arcs in the net).

To illustrate *CO-OPN/2*, we introduce an example that will be used through this paper: a phonecard system. This example models the behavior of a telephone machine that can be used with a phonecard. We model this system using several *ADTs* (Pin, Money, Bool and Nat) and classes (PhoneCard and Telephone).

```
Class PhoneCard;
Interface
  Use Pin, Money;
  Type phonecard;
  Creation create _ : pin;
  Methods
    get-pin _ : pin;
    withdraw _ ,
    get-balance _ : money;
Body
  Places
    balance : money;
    id : pin;
  Initial
    balance 20;
  Axioms
    create p : -> id p;
    get-pin p : id p -> id p;
    get-balance b : balance b -> balance b;
    (b >= m) = true => withdraw m :
          balance b -> balance b - m;
  Where b, m:money; p:pin;
End PhoneCard;
```

```
Class Telephone;
Interface
  Use Pin, Money, PhoneCard, Booleans;
  Type telephone; Object cabin : telephone;
  Creation create;
  Methods
    insert _ : phonecard;
    enter _ : pin;
    buy _ : money;
Body
  Places
    idle : money;
    wait-for-pin: phonecard money;
    wait-for-buy: phonecard money;
    ready-to-eject: phonecard money;
  Initial idle 0;
  Transition eject;
  Axioms
    insert c: idle s -> wait-for-pin c s;
    (pp = p) = true => enter p With c.get-pin pp :
          wait-for-pin c s -> wait-for-buy c s;
    (pp = p) = false => enter p With c.get-pin pp :
          wait-for-pin c s -> ready-to-eject c s;
    (m >= b) = true => buy m With c.get-balance b :
          wait-for-buy c s -> ready-to-eject c s;
    (m >= b) = false => buy m
       With c.get-balance b .. c.withdraw m:
          wait-for-buy c s -> ready-to-eject c s+m;
    eject: ready-to-eject c s -> idle s;
  Where s, m, b: money; c: phonecard; p, pp : pin;
End Telephone;
```

Figure 1. Textual specification of the classes Phonecard and Telephone

The figure 1 shows the textual description of the class Phonecard. The state of a phonecard is described by the place balance, which stores the money available on the card, and id, which stores the pin-code. The balance is initialized (keyword **Initial**) to a constant value 20 for each new card. Four methods are exported by this class: to create the phonecard (create), to get the pin-code (get-pin), to access the balance (get-balance), and to reduce it (withdraw). In the field **Axioms**, the behavior of the

methods is given by the behavioral axioms described above. The class Telephone specifies the behavior of the automaton which accepts a card, waits for and checks a pin-code, and, as long as the pin-code is correct, reduces the balance of the card of a given amount corresponding to the price of a phone call.
The semantics of a *CO-OPN/2* specification is based on labeled transition systems. These rules are expressed as *structured operational semantics* and they build a deduction system over the axioms. The semantics expressed by the rules is that the behavior of a set of objects is calculated by starting from the lowest object in the hierarchy and repeatedly adding a new object to the system. We may thus build the graph of all the possible behaviors of a specification, and build proof trees, which allow both selecting cases of satisfaction and non satisfaction for *CO-OPN/2* formulae containing variables, and validating test cases, for ground *CO-OPN/2* formulae.

3.2 The Theory of Testing

The theory of testing is elaborated on specifications *SPEC*, programs *PROG* and test cases *TEST*, and on adequate compatible satisfaction relationships between programs and specifications, $\models$, and between programs and test cases, $\models_O$. This is defined by the following equation:

$$(P \models_O T_{SP} \Leftrightarrow P \models SP).$$

The equivalence relationship $\Leftrightarrow$ is satisfied when the test set T_{SP} is pertinent, i.e. valid (any incorrect program is discarded) and unbiased (it rejects no correct program). However, a pertinent test set T_{SP} can only be used to test a program P if T_{SP} has a "reasonable" finite size. Limiting the size of a test sets is performed by sampling. In our theory, sampling is performed by applying hypotheses on P, i.e. by making assumptions that the program reacts in the same way for some inputs.

Expressing Test Cases with HML Formulae

For *CO-OPN/2*, the test cases can be expressed with the *HML* Logic introduced by Hennessy-Milner in [Hennessy and Milner, 1985][1]. *HML* formulae built using the operators *Next* (<_>), *And* ($\wedge$), *Not* ($\neg$), *T* (always true constant), and the events *EVENT* (*SP*) of the specification $SP \in SPEC$, are noted HML_{SP}. An advantage of this approach is to have an observational description of the valid implementation through the test cases. A test case is a formula which is valid or not in the specification, and which must be experimented in the program, i.e. a correct implementation behaves similarly on the test cases.

An elementary test case for a program under test $P \in PROG$ and a specification $SP \in SPEC$ can be defined as a couple <*Formula*, *Result*> where:

- *Formula* $\in HML_{SP}$: (ground) temporal logic formula.
- *Result* $\in$ {*true*, *false*}: boolean value showing whether the expected result of the evaluation of *Formula* (from a given initial state) is *true* or *false*.

[1] It exists a full agreement between the bisimulation equivalence and the HML^{∞} equivalence (HML^{∞} is *HML* with the infinite conjunction (see [Hennessy and Stirling, 1985])). The bisimulation equivalence identifies correct implementations with respect to the specifications. Bisimulation is stronger than other equivalence relationships, such as some of those proposed in [de Nicola and Hennessy, 1984], in that it assumes that non specified behaviors are not acceptable for the implementation.

A test case <*Formula, Result*> is successful if *Result* reflects the validity of *Formula* in the labeled transition system modeling the expected behavior of *P*, for instance:

T1: <<c.create 1234> (<c.withdraw 20> <c.get-balance 0>*T* and <c.get-pin 1234>*T*), true>

In all other cases, a test case <*Formula, Result*> is a fail. It is important to note that the test case definition will allow the test procedure to verify that a non-acceptable scenario cannot be produced by the program (for instance, to make a call even though the identification code of the phonecard is wrong), i.e.

T2: <<c.create 1234> <cabin.insert c> <cabin.enter 5629> <cabin.buy 5>*T*, false>

For test cases expressed using HML_{SP}, we can define the exhaustive test set $EXHAUST_{SP, Ho} \subseteq TEST$, which includes all possible behaviors derivable from the specification, as:

$EXHAUST_{SP, Ho} = \{ <Formula, Result> \in HML_{SP} \times \{true, false\} \mid (SP \models_{HML_{SP}} Formula \text{ and } Result = true) \text{ or } (SP \not\models_{HML_{SP}} Formula \text{ and } Result = false)\}$.

Practicable Test Context and Hypotheses

Assuming that hypotheses *H* have been made on *P*, the following formula has to be verified for any selected test sets $T_{SP, H}$:

$$(P \text{ satisfies } H) \Rightarrow (P \models_O T_{SP, H} \Leftrightarrow P \models SP).$$

Thus, the test selection problem is reduced to applying hypotheses to a program until a test set of reasonable size is selected. For that purpose, we build a test context, called practicable because it is pertinent and can be effectively applied to the oracle. Given a specification *SP*, a practicable test context $(H, T)_O$ is defined by a set of hypotheses *H* on a program under test *P*, a test set *T* of "reasonable" finite size and an oracle *O* defined for each element of *T*. The selection of a practicable test set *T* is made by successive refinements of an initial test context $(H_0, T_0)_O$ which has a pertinent test set T_0 (possibly of not "reasonable" size), until obtaining a practicable test context $(H, T)_O$: $(H_0, T_0)_O \leq \ldots (H_i, T_i)_O \leq (H_j, T_j)_O \ldots \leq (H, T)_O$.

At each step, the preorder refinement context $(H_i, T_i)_O \leq (H_j, T_j)_O$ is such that:

- The hypotheses H_j are stronger than the hypotheses H_i
- The test set T^j_{SP, H_j} is included in the test set T^i_{SP, H_i}
- If *P* satisfies H_j then $(H_j, T_j)_O$ detects no more errors than $(H_i, T_i)_O$
- If *P* satisfies H_j then $(H_j, T_j)_O$ detects as many errors than $(H_i, T_i)_O$

Therefore, if T^i is pertinent then T^j is pertinent. Since the exhaustive test set is pertinent, we can use it for the initial context T^0.

Test Selection

From a practical point of view, the reduction process is implemented as a selection process: to each reduction hypothesis on the program corresponds a constraint on the test set. Indeed, the exhaustive test set can be defined as a couple $< f, r >$ where *f* is a HML_{SP} formula with variables universally quantified. The aim of the test selection becomes the reduction of the level of abstraction of *f* by constraining the instantiation of its variables. This will be presented in details in section 5.

For that purpose we define *Var (f)*, the set of variables in the formula *f*, and we introduce the language HML_{SP} with variables, $HML_{SP, X}$, build using the standard *HML* operators, the events with variables $EVENT\ (SP, X_S)$ of the specification $SP \in$

SPEC, and variables. The variables of $HML_{SP, X}$ belong to $X = X_{HML} \cup X_{event} \cup X_S$ where:

- X_{HML}: variables of type $HML_{SP, X}$ formula,
- X_{event}: variables of type event,
- $X_S = X_{adt} \cup X_c$: variables of type ADT and class (reference to objects).

For instance, in the telephone example, the $HML_{SP, X}$ formula f = <cabin.insert (o)> <cabin.enter (p)> <e> g has the variables g $\in X_{HML}$, e $\in X_{event}$, c, o $\in X_c$, and p $\in X_{adt}$. We note HML_{SP, X_S} the $HML_{SP,X}$ language in which X is restricted to X_S.

To replace those variables by values, we use interpretations (the set of all the interpretations is called *INTER*). To replace those variables by terms, we use substitutions (the set of all the substitutions is called *SUBS*). The evaluation of a term in the domain D is performed using the function $[\![\ ]\!]_D$: term-D -> D. We also define the concatenation $f \mid g$ of a HML_{SP, X_S} formula f and a $HML_{SP, X}$ formula g as the $HML_{SP,X}$ formula obtained by replacing all T in f by g.

The Oracle

Once a test set has been selected, its elements are executed on the program under test. The choice of the *HML* logic to express test cases allows executing easily those test cases. A logic with more complex modalities could have led to nonexecutable test cases. Then the results collected from this execution are analyzed. It is the role of the oracle to perform the analysis, i.e. to decide the success or the failure of the test set. The oracle O is a partial decision predicate of a formula in a program P. Since oracles are not always able to compare all the necessary elements to determine the success or the failure of a test we introduce oracle hypotheses H_O as part of the possible hypotheses and collect all power limiting constraints imposed by the realization of the oracle. More information about the construction of oracles for our approach is given in [Barbey et al., 1996] and [Barbey, 1997].

4 THE TEST SELECTION PROCESS

The previous description of the test selection process was mainly concerned with the theoretical justification of the approach correctness. In the next sections we will emphasize the practical problems that appear when practical test sets have to be produced. The test selection process is performed in the following steps:

- Focus on a particular unit (class) of interest to test in details, the class under test. This unit must be an independent unit (which does not use any other unit), or a unit which uses other units **already tested** or replaced by **stubs**.The introduction of stubs should be avoided by introducing a test order that allows the integration of already tested components. This minimizes the testing effort and focus the test process on successive enrichments of the specification. The class under test will be tested through one of its instances. Possibly more instances of the class can appear if needed in methods parameters.
- Deduce the test environment from the focus: The test environment is the set of all the units used (directly and indirectly) by the focus.
- Define a system of constraints on the exhaustive test set with the help of reduction hypotheses as follows:

- For the focus: use 'weak' reduction hypotheses (like regularity, see section 5.2) to preserve as much as possible the quality of the test set.
- For the other units: use 'strong' reduction hypotheses (like uniformity, see section 5.3) to minimize as much as possible the size of the test set. Uniformity hypotheses can be used on subdomains, which implies the computation of the variables' subdomains of validity (following a given criteria of test coverage) by unfolding techniques (see section 6).

- Solve the system of constraints previously defined, based on the inference rules of *CO-OPN/2*. This results in a test set of 'reasonable' size.

Of course, to select a complete test set for a given class, it may be necessary to define several systems of constraints that exercise different aspects of the specification. Throughout the test process, the test cases can be validated by computation of the value of the variable *Result*.

Because of the definition of a test as a couple $<HML_{SP}$ *formula, Result*$>$, and because of its construction's mechanism, a test set could contain redundant test cases. A redundant test is a test which can be suppressed from the test set without altering its pertinence (validity and unbiais). For instance, the test cases $<f$, *true*$>$ and $<\neg f$, *false*$>$ are redundant, as well as the test cases $<f \wedge g$, *true*$>$ and $<g \wedge f$, *true*$>$. To eliminate such structural redundancies, a test set can be transformed into a test set free of redundant test cases, called a *minimal test set.* This transformation can occur at any time during the process when it is possible to perform it.

5 HYPOTHESES AND STRATEGIES FOR TEST SELECTION

It is often reasonable to start the test set selection process by applying regularity hypotheses on the program, i.e. by constraining the $HML_{SP, X}$ formulae. Then uniformity hypotheses can be applied, i.e. the instantiation of the remaining variables of the $HML_{SP, X}$ formulae can be constrained. This section presents these different reduction hypotheses and constraints, and the associated strategies used in practice. Note that, as mentioned in [Arnould, 1997], the strategies need to be carefully built to implement the corresponding hypotheses. The danger is to obtain test sets which do not keep the pertinence of the initial exhaustive test set.

5.1 Language of Constraints

The constraints are expressed in a language called $CONSTRAINT_{SP, X}$, which includes the set of all the constraints applicable on $HML_{SP, X}$. The syntax and the semantics of $CONSTRAINT_{SP, X}$, are defined by means of elementary constraints applicable either on the events or on the shape of the $HML_{SP, X}$ formulae. The elementary constraints are built using numeric functions like *nb-events or nb-occurrences*, boolean functions like *only* or *shape* and *HML* functions like | *(concatenation)*. The satisfaction relationship $\models^{I}_{C} \subseteq CONSTRAINT_{SP}$ is partially defined as follows:

$\models^{I}_{C}$ $(C_1 \wedge C_2) \Leftrightarrow (\models^{I}_{C} C_1 \wedge \models^{I}_{C} C_2)$

$\models^{I}_{C}$ $(nb\text{-}events\ (f) = k) \Leftrightarrow (\llbracket nb\text{-}events\ (f) \rrbracket_{IN} = \llbracket k \rrbracket_{IN})$

$\models^{I}_{C}$ $(nb\text{-}events\ (f) = x) \Leftrightarrow (\llbracket nb\text{-}events\ (f) \rrbracket_{IN} = \mathbf{I}\ (x))$

where $f \in HML_{SP, X_S}$, $k \in IN$, x is a variable of type IN and $\mathbf{I} \in INTER$

Other constraints will be presented in the following text. The complete definition of $CONSTRAINT_{SP, X}$ can be found in [Barbey, 1997].

5.2 Regularity Hypotheses

The regularity hypotheses stipulate that if a test $<f, r>$ in which the formula f contains a variable v, is successful for all instances of v satisfying a constraint C, then it is successful for all possible instances of v. For instance, in the case of the telephone system and a constraint *"the number of insertions of a phonecard is equal to 20"*, if all test cases $<f, r>$, in which the constraint is satisfied, are successful, then the system behaves correctly for all possible numbers of insertions.

Definition: Regularity hypothesis
Given a specification $SP \in SPEC$, a test $<f, r> \in HML_{SP, X} \times \{true, false\}$, a variable $v \in Var\ (f)$, a constraint $C \in CONSTRAINT_{SP, X}$. A regularity hypothesis of constraint C on a variable v for a test $< f, r >$ and a program P, is such that:

$$\forall\, r \in \{true, false\},\ ((\forall\, (v_0 / v) \in SUBS,\ (\forall\, \mathbf{I}_0 \in INTER,\ (\models_C^{\mathbf{I_0}}\ (v_0 / v)\ (C) \Rightarrow P \models_o <\mathbf{I}_0((v_0 / v)\ (f)), r>))) \Rightarrow (\forall\, (v_1 / v) \in SUBS,\ (\forall \mathbf{I}_1 \in INTER, (P \models_o <\mathbf{I}_1((v_1 / v)\ (f)), r>)))).$$

This definition means that if for all substitutions (v_0 / v) the satisfaction of the constraint C implies that P satisfies the formula f in which v is replaced by v_0, then for all substitutions (v_1 / v) P will satisfy the formula f in which v is replaced by v_1. The role of the two interpretations $\mathbf{I}_0$ and $\mathbf{I}_1$ is just to replace the remaining variables by values, to deal with ground constraints and formulae during the evaluations. For instance, if the constraint C and the substitution (v_0 / v) force the *HML* formula f to have the structure: $f = <e>\ g$ where e is an event and g a variable of type *HML*, g must be replaced by all its possible interpretations $\mathbf{I}_0$ to obtain ground formulae which could be evaluated. For instance:

$\mathbf{I}_{0,\alpha}(g) = <e>\ \mathsf{T} \Rightarrow f = <e>\ <e>\ \mathsf{T}$,
$\mathbf{I}_{0,\beta}(g) = \text{not } <e>\ \mathsf{T} \Rightarrow f = <e>\ \text{not } <e>\ \mathsf{T}$,...

To each regularity hypothesis on the program is associated a predicate and a strategy. According to the former definition, the predicate is a constraint $C \in CONSTRAINT_{SP, X}$ and the strategy aims at finding the test cases which satisfy this constraint.

Regularity on Events

This section gives some examples of hypotheses having constraints on the events of the HML_{SP,X_S} formulae.

a. Number of Events
Hypothesis: If a test $<f, r>$ is successful for all instances of f having a number of events equal to a bound k, then it is successful for all possible instances of f. The number of events is computed recursively with the function *nb-events* as follows:

Definition: Semantics of *nb-events*: $HML_{SP, X_S} \rightarrow IN$
- *nb-events* $(T) = 0$
- *nb-events* $(\neg f)$ = *nb-events* (f)
- *nb-events* $(f \wedge g)$ = *nb-events* (f) + *nb-events* (g)

• *nb-events* (<*e*> *f*) = *nb-events* (*f*) + 1

where $e \in$ *EVENT* (*SP*, X_S).

Thus the constraint $C \in CONSTRAINT_{SP, X}$ is the predicate: *nb-events* (*f*) = *k*.

Strategy: The strategy used to solve the former constraint *C* generates all the HML_{SP, X_S} formulae with a number of events equal to *k*, without redundancy. With this strategy, only skeletons are generated and nothing is imposed by the specification. Later, free variables will be instantiated to events based on methods of the environment. For instance, the constraint *nb-events* (*f*) = 2 produces the four following test cases (where the variable V_0, and V_1 are of type event):

T0: <(not <V_0> T) and (not <V_1> T), *result*>

T1: <(not <V_0> T) and (<V_1> T), *result*>

T2: <(<V_0> T) and (<V_1> T), *result*>

T3: <<V_0> <V_1> T, *result*>

b. Occurrences of a Given Method

Another way to reduce the size of the test sets is to constrain the number of occurrences of a given method in each test.

Hypothesis: If a test <*f*, *r*> is successful for all instances of *f* having a number of occurrences of a given method *m* equal to a bound *k*, then it is successful for all possible instances of *f*.

The number of occurrences of a given method *m* is recursively computed with the function *nb-occurrences*: $HML_{SP, X_S} \times METHOD \rightarrow I\!N$, which is defined like the function *nb-events*. The constraint $C \in CONSTRAINT_{SP, X}$ is the predicate *nb-occurrences* (*f*, *m*) = *k*.

Strategy: The strategy used to solve the former constraint *C* generates all the HML_{SP, X_S} formulae with a number of events based on the method *m* equal to *k*. For instance, for a phonecard c we can do the assumption:

nb-occurrences (*f*, get-balance) = 1 -- 1 occurrence of get-balance

which leads to this kind of test cases:

T: <<c.create V_0> <c.get-balance V_1> <c.withdraw V_2> T, *result*>

where the variables V_0, V_1, V_2 are of type ADT.

c. Event Classification

The events used in the test cases can be based on the kinds of the methods of the environment. The kinds of actions that are performed by the events are classified into *creators* (creators allow to create the state of the objects), *modifiers* (modifiers allow to modify the state of the objects) and *observers* (observers allow to observe the state of the objects but not to modify it). For instance, in the class Phonecard, the events based on create are creators, the events based on withdraw are modifiers and the events based on get-pin and get-balance are observers.

Hypothesis: If a test <*f*, *r*> is successful for all instances of *f* which are a combination of creators followed by a combination of modifiers and terminated by a combination of observers, then it is successful for all possible instances of *f*.

Thus the constraint $C \in CONSTRAINT_{SP, X}$ is the following:

$(f = (f_c \mid (f_m \mid f_o))) \wedge only_{creator}(f_c) \wedge only_{modifier}(f_m) \wedge only_{observer}(f_o)$,

where the function $only_i$: $HML_{SP, X_S} \rightarrow \{true, false\}$ (i = {*creator*, *modifier*, *observer*}) is recursively defined in a fashion similar to the function *nb-events*.

Strategy: The strategy used to solve the former constraint C generates all the HML_{SP, X_S} formulae which are a combination of creators (used to create the objects of the system) followed by a combination of modifiers (used to describe state evolution) and terminated by a combination of observers. For instance using this strategy on a phonecard c with the creator create, the modifier withdraw and the observers get-pin and get-balance, allows to generate this kind of test:

<<c.create V_0> <c.withdraw V_1> <c.get-balance V_2> T, *result*>

where the variables V_0, V_1, V_2 are of type ADT.

d. State Classification

We can divide the methods of a class in sets according to the places that are connected to them. If there is no interaction between those sets in the specification and in the implementation, testing those interactions will not help much in discovering errors. This division can be based on the pre- and post conditions of the axioms, which define how events read and modify the contents of places.

For example, the methods of PhoneCard can be divided in two sets, according to whether they are based on balance or on id: the method get-pin is involved in the security of the card and is connected to the place id, while the methods get-balance and withdraw are involved in the accounting of the card, and are connected to the place balance (see figure 2).

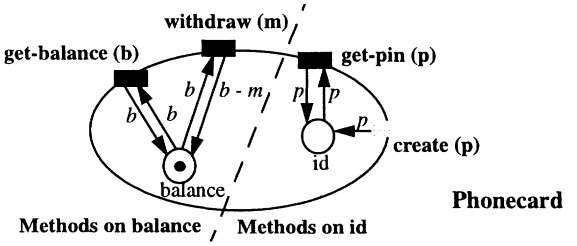

Figure 2. Classification of methods wrt to their state of interest

Hypothesis: Given M (P_i) the set of methods applied to the set of places P_i such that $\forall$ i, j, i $\neq$ j, $P_i \cap P_j = \emptyset$, a formula f_i containing only methods included in M (P_i) will not uncover more errors in the methods of M (P_i) than a formula f_j made of f_i interleaved with methods of M (P_j), i $\neq$ j. Thus, if a test set for M (P_i) and another test set for M (P_j) are successful, all test cases for M ($P_i \cup P_j$) are successful too.

In the above example, we can make the hypothesis that the behavior of M ({balance}), i.e. withdraw and get-balance, is independent from that of M ({id}), i.e. get-pin, and that these methods need not be tested in interaction (i.e. we need not generate test cases for M ({balance, id})). Thus, the constraint $C \in CONSTRAINT_{SP, X}$ is the predicate: *only* (f, *Methods* (P_i)) = *true* where the function *only*: $HML_{SP, X_S} \times \mathcal{P}$ (METHOD) $\rightarrow$ {*true, false*} is recursively defined akin to the function *nb-events*.

Strategy: The strategy used to apply the former constraint C generates only $HML_{SP, X}$ formulae that are bound to a particular place or set of places. For instance, applying the constraint on the place balance, i.e. $(f = (f_c \mid f_b)) \wedge only_{creator}(f_c) \wedge only(f_b, \{withdraw, get\text{-}balance\}) = true$, will provide test cases like

<<c.create V_0> <c.withdraw V_1> <c.get-balance V_2> <c.withdraw V_3> T, result>

This hypothesis is strong, because it assumes that the program follows the specification, in that the lack of interaction in the specification is conveyed in the program.

However, in some cases, it can be shown by a static analysis of the program or simply by a code review, assuming that there is a simple morphism between the places in the specification and the class components in the program. The algorithm to find the sets of methods must take into account connected objects.

Regularity on the Shape of HML Formulae

This section presents a constraint applicable on the shape of the $HML_{SP, X}$ formulae, i.e. a constraint allowing to force the shape of the test cases.

Hypothesis: If a test case $<f, r>$ is successful for all instances of f having a given shape s, then it is successful for all possible instances of f. The formulae f of shape s are detected using the function *shape:* $HML_{SP, X} \times HML_{SP, X} \rightarrow \{true, false\}$ recursively defined like the function *nb-events*. The constraint $C \in CONSTRAINT_{SP, X}$ is the predicate: *shape* $(f, s) = true$

Strategy: The strategy used to solve the constraint $C = (shape\ (f, s) = true)$ generates all the $HML_{SP, X}$ formulae f of shape s. For instance, for a phonecard c with the pin-code 1234 and the initial balance 20, we can express the following constraint:

shape (f, <c.create 1234> ($f_0 \wedge \neg$ <c.get-balance 25> f_1))*=true*

which leads to test cases of the shape:

<<c.create 1234> (f_0 and not <c.get-balance 25> f_1), result>

in which the two variables f_0 and f_1 are $HML_{SP, X}$ formulae.

This section shows that the combinatorial explosion can be reduced by constraining the shape of the $HML_{SP, X}$ formulae. Obviously, we can imagine a lot of other constraints of this type, for instance "*$HML_{SP, X}$ formulae with a given number of 'and' operators*" or "*$HML_{SP, X}$ formulae with a given number of 'not' operators*", ...

Choosing Regularity Hypotheses

During the test selection process, the tester selects regularity hypotheses with respect to his own knowledge of the type of faults that can occur in the program. For instance, this knowledge can be derived from a graphical representation of the specification, which allows an intuitive comprehension and an easy understanding of the behavior of each unit of the system. This is very helpful for hypotheses like *nb-events, nb-occurences, shape*. For instance, for the Telephone, a phone call always begins by the insertion of a phonecard (method insert) followed by the entrance of the pin-code (method enter). Thus for the test of the class Telephone, the tester can choose to have test cases satisfying the constraint:

shape (f, <c.create 1234> <cabin.insert c> <cabin.enter p> *g) = true*

and test cases satisfying the constraint

shape (f, <c.create 3961> <cabin.buy m> *g) = true.*

This second test set will only contain test cases that verify that the implementation does not allow events to happen if the user omits to insert his phonecard.

Other hypotheses less dependent of the tester's knowledge of the specification, like "*event classification*" and "*state classification*", can be applied systematically.

5.3 Uniformity Hypotheses

The application of constraints on the exhaustive test set generates test cases with variables. These variables can be replaced using various strategies, like *exhaustive-*

ness or *uniformity*. Exhaustiveness implies that each variable is replaced by all its possible instances. It can be useful, but it can lead to a test set infinite or having an 'unreasonable' size. To overcome this problem, uniformity hypotheses can be used.

The uniformity hypotheses stipulate that if a test case $<f, r>$ in which the formula f contains a variable v, is successful for a given value of v, then it is successful for all possible values of v. Thus uniformity hypotheses are performed to limit the test cases selected for the variables in a formula f by selecting a unique instance of each variable v in $Var\ (f)$.

Definition: Uniformity hypothesis
Given a specification $SP \in SPEC$, a test case $<f, r> \in HML_{SP,\ X} \times \{true, false\}$, a variable $v \in Var\ (f)$. An uniformity hypothesis on a variable v for a test case $<f, r>$ and a program P, is such that:
$\forall\ r \in \{true, false\}, \forall\ (v_0 / v) \in SUBS,$
$((P \vDash_o <(v_0 / v)\ (f), r>) \Rightarrow (\forall\ (v_1 / v) \in SUBS, P \vDash_o <(v_1 / v)\ (f), r>)).$

This definition means that for all result r of $\{true, false\}$ and for all substitution (v_0 / v) we have: if P satisfies the formula f in which v is replaced by v_0, then for all substitution (v_1 / v) P will satisfied the formula f in which v is replaced by v_1.

Four kinds of variables can occur in a formula f: $HML_{SP,\ X}$ formulae, events, objects (class instances), and algebraic values (ADT instances). The strategy for uniformity of the four kinds of variables is the following:

- uniformity on $HML_{SP,\ X}$ formulae: Any $HML_{SP,\ X}$ formula can be randomly selected, with respect to the constraints applied on the enclosing test formula.
- uniformity on events: Any event can be randomly selected, with respect to the constraint that it is applied to an object in the focus environment.
- uniformity on algebraic values: Any algebraic value can be generated by applying a random combination of the functions defined in the corresponding ADT.
- uniformity on objects: An object in any state can be generated. Generating randomly a sequence of events including first a creation method, and then a (possibly empty) sequence of modifiers produce such object. The observers s are not relevant This sequence must be injected in the formula in construction at a place before the event containing the variable to which the uniformity is applied.

Uniformity hypotheses are very strong, because the coverage of the signature of methods is weak for the domain of the units under test. Therefore, they are usually not applied to the component under test, but to the components imported into the specification, which we suppose being already tested. Uniformities can only be applied with a satisfying coverage when the semantics of the operation include no calculation based on the variable on which the hypothesis is applied — e.g. the observer methods — or if the operation only considers the reference to the object, and not its state — e.g. the stored item in most container classes—. In some cases, a static analysis of the program can show the validity of a uniformity hypothesis by examining the use of the object on which the uniformity hypotheses are applied.

6 UNIFORMITY AND SUBDOMAIN DECOMPOSITION

The coverage of the tested unit can be very low when applying a uniformity hypothesis if, by selecting an unique instance, cases of the specification are not covered.

For example, the method enter of the class Telephone specifies a different behavior depending on the condition (pp = p): if the condition is true, the telephone will be ready to accept a call, if not, it will eject the card. For a good coverage of the method enter, test cases must be performed to verify the behavior by introducing a valid code ((pp = p) = true) and an invalid code ((pp = p) = false). A uniformity on the parameter p of the method enter will only select one value of pp, and will miss one of the two specified behaviors. Thus, applying a uniformity hypothesis on pp will result in not covering all specified behaviors, leading to a test set of low quality.

The strategy to obtain a good coverage of a formula f whenever constraints exist on the domain of any free variable v of f, is to apply uniformity hypotheses with subdomain decomposition. The subdomains for the different variables are considered as constraints, and are handled in conjunction to form sets of constraints on formulae on which uniformity hypotheses are then applied to select ground formulae. For a formula f with free variables $Var_{X_S}(f)$, it consists in finding the possible behaviors encompassed in f, and, for each found behavior, to select a test case in which the variables in $Var_{X_S}(f)$ are assigned values that will cause this behavior. In *CO-OPN/2*, variables with a domain belong to X_S, i.e. to algebraic values (of ADTs) and objects (of classes), excluding variables in X_{Event} and X_{HML}. It is difficult to analyze the possible behaviors and consequently to perform subdomain decomposition on a formula if the event and *HML* variables have not been fixed.

Finding the behaviors of a formula f is performed by enumerating the *valid* and *invalid* sequences in the transition system of the specification under test and to find the constraints on the elements of $Var_{X_S}(f)$ that will result in the execution of these sequences. Thus, uniformity subdomains are *sets of constraints on the variables of a formula f*. The union of all the uniformity subdomains should include all the possible sequences of transition derivable from the axioms of the events in the formula.

Subdomain decomposition on a formula f is a three steps process:

Step 1: **Derivation of constraints from the specification:** Find out the possible behaviors of the formula f, i.e. the constraints on the variables in f found in the behavioral axioms.

Step 2: **Computation of subdomains from the constraints:** Unfold the constraints to find uniformity subdomains on algebraic axioms.

Step 3: **Test set selection from the subdomains:** For each uniformity subdomain, solve the constraints and select values for the variables.

6.1 Derivation of constraints from the specification

The constraints can be divided in two groups: β-constraints and σ-constraints.

- β-constraints — behavioral constraints — drive the possible executions. They can influence the ability to trigger an event, and are algebraic or synchronization conditions. β-constraints are found by performing a case analysis of the behavioral axioms. Generally, we will discern two choices for each construct: the case of a success (i.e. valid behavior), and the case of failure (invalid behavior).
- σ-constraints — substitution constraints — are constraints that must hold between variables, given the β-constraints. σ-constraints make up the "glue" between the different events in a formula. The constraints are equalities between the variables in the derivation trees.

The β-constraints are found in the constructs of the behavioral axioms:

- Algebraic condition and method parameters on an axiom limit its domain of validity. Since our approach also deals with failures, two β-constraints are drawn up, in concordance with the results of the condition. We include a β-constraint for which the condition yields true and another one for which it yields false.
- β-constraints can be drawn up from synchronization expressions by enumerating the possible synchronization cases.
- Pre- and postconditions give information on what the state of an object will be after triggering the event, assuming that the initial state (i.e. before triggering) satisfies the precondition. Thus, they are helpful in selecting β-constraints based on the preconditions, and will also provide σ-constraints based on an analysis of the relationships between the initial state and the preconditions.

The semantics of an event can be described with several behavioral axioms (Axiom$_i$). When enumerating the possible sequences in the transition system, this leads to as many possible choices for a given method as axioms for this method, unless the axioms cover each other, in which case the system is not deterministic, and it may not be possible to ensure the coverage of all axioms.

Therefore, this first step can be divided in two sub-steps:

Step 1.1: decide upon β-constraints for each event in the formula. To each β-constraint corresponds a possible (valid or invalid) execution. Since we are not only interested in selecting test cases in the domains of validity of the formula, but also in its possible failures, we will not only consider the cases where the constraints are satisfied, but also the cases when the constraints are not satisfied.

Step 1.2: calculate the σ-constraints corresponding to β-constraints. These constraints can be calculated from the derivation tree deducted from the inference rules, given the β-constraints.

6.2 Computation of subdomains from the constraints

The second step in the strategy is to unfold the constraints to extract subdomains. The constraints must be solved in conjunction using logical properties such as transitivity to take into account the entire formula and the properties of the axioms. Two cases can occur:

- The constraint is an algebraic condition. In this case, unfolding is performed as described in [Bernot et al., 1991].
- The constraint involves objects. In this case, a formula must possibly be generated to select an object that satisfies the constraint.

Since β-constraints express behaviors that must hold, these constraints are only unfolded in their domain of validity, and must remain in the constraints. σ-constraints however express basic substitutions, that occur at a single step when calculating the derivation trees, and thus may disappear from the constraints. To find out invalid behaviors, σ-constraints will also be unfolded outside their domains of validity, which means, since they are equalities, that we will consider uniformity subdomains with the corresponding inequality.

Of course, when unfolding constraints, several uniformity domains can appear given the number of variables appearing in the constraint.

6.3 Test set selection from the subdomains

The third step in the strategy is to solve constraints and to select values. For each subdomain, values that satisfy the conditions in the uniformity subdomain must be assigned to the variables. Uniformity subdomains for which the constraints cannot be solved will be dropped. For each free variable, a single instance uniformity hypothesis is applied per uniformity subdomain.

6.4 Example

Given the following formula, which includes an *HML And* operator, and satisfies the constraint $(f=((f_c \mid f_m) \mid f_o) \wedge only_{creator}(f_c) \wedge only_{modifier}(f_m) \wedge only_{observer}(f_o)) = true$):
f= <c.create V_0> <c.withdraw V_1>(<c.get-pin V_2> and <c.get-balance V_3>)T

Step 1.1. The only constraint is on the method withdraw, from which we extract two β-constraints, β_1 and β_2, corresponding to the case of the balance bigger than the withdrawn money, and the opposite case. We decompose f in two sequences of events. This decomposition is equivalent to the original formula, and makes calculations simpler because of the determinism of the sequence R1 (<c.create V_0> <c.withdraw V_1>). We show on figure 3 the inference trees for β_1.

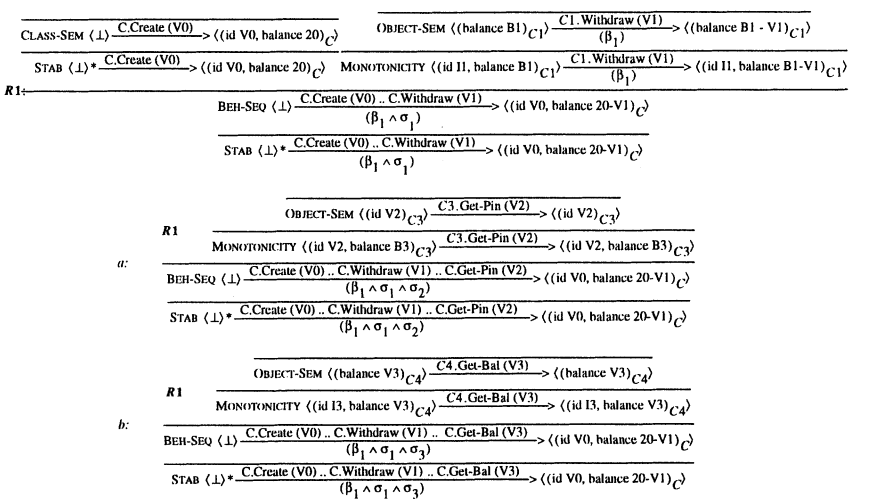

Figure 3. Inference trees for β_1

- *a:* <c.create V_0> <c.withdraw V_1> <c.get-pin V_2> T
- *b:* <c.create V_0> <c.withdraw V_1> <c.get-balance V_3> T.

Step 1.2. From the derivation trees for β_1 and β_2 (which is not shown), we can extract the set of subdomains S.

$$S = \left\{ \begin{array}{l} \langle \beta_1 = (B1 \geq V1), \sigma = \sigma_1 \wedge \sigma_2 \wedge \sigma_3 \rangle, \\ \\ \langle \beta_2 = \neg(B1 \geq V1), \sigma = \begin{pmatrix} c = c1 \\ V0 = I1 \\ 20 = B1 \end{pmatrix} \rangle \end{array} \right\} \quad \sigma_1 = \begin{pmatrix} c = c1 \\ V0 = I1 \\ 20 = B1 \end{pmatrix}, \sigma_2 = \begin{pmatrix} c2 = c3 \\ V2 = I2 \\ B2 = V3 \end{pmatrix}, \sigma_3 = \begin{pmatrix} c = c2 \\ V0 = V2 \\ V3 = 20 - V1 \end{pmatrix}$$

We drop the constraints on the reference ($c = c1 = c2 = c3$) because they are always true. From the first tuple of S, we extract that for $(20 \geq V_1)$ = true, $(V_0 = V_2)$ = true and $(V_3 = 20\text{-}V_1)$ = true, and from the second tuple, that for $(20 \geq V_1)$ = false, Result will always be false, whatever the values of V_0 and V_2 are.

Step 2. For the first member of S, we unfold the second and the third conditions into true and false equalities. This leads to four uniformity subdomains:

1: <{V_0, V_1, V_2, V_3}, {$(20 \geq V_1)$ = true, $(V_0 = V_2)$ = true, $(20\text{–}V_1 = V_3)$ = true}, *f*, true>

2: <{V_0, V_1, V_2, V_3}, {$(20 \geq V_1)$ = true, $(V_0 = V_2)$ = true, $(20\text{–}V_1 = V_3)$ = false}, *f*, false>

3: <{V_0, V_1, V_2, V_3}, {$(20 \geq V_1)$ = true, $(V_0 = V_2)$ = false, $(20\text{–}V_1 = V_3)$ = true}, *f*, false>

4: <{V_0, V_1, V_2, V_3}, {$(20 \geq V_1)$ = true, $(V_0 = V_2)$ = false, $(20\text{–}V_1 = V_3)$ = false}, *f*, false>

No unfolding is performed on the second member. This leads to one additional uniformity subdomain:

5: <{V_0, V_1, V_2, V_3}, {$(20 \geq V_1)$ = false}, *f*, false>

Step 3. After selecting values by applying uniformity hypotheses on each subdomain, we get the following test cases:

1: <<c.create 1234> <c.withdraw 12> (<c.get-pin 1234> T and <c.get-balance 8> T), true>

2: <<c.create 4123> <c.withdraw 6> (<c.get-pin 4123> T and <c.get-balance 5> T), false>

3: <<c.create 111> <c.withdraw 17> (<c.get-pin 1234> T and <c.get-balance 3> T), false>

4: <<c.create 45> <c.withdraw 17> (<c.get-pin 5371> T and <c.get-balance 6> T), false>

5: <<c.create 53> <c.withdraw 50> (<c.get-pin 53873> T and <c.get-balance 6> T), false>

7 CONCLUSION

In [Barbey et al., 1996], we have presented a generalization and an adaptation of the Bernot, Gaudel, and Marre theory of testing to object oriented software. In this paper, we presented the methodological grounds for applying this theory in practice. We have described a practical process for test selection, based on the construction of constraints. First, we have presented several regularity hypotheses, including the strategies to put them into practice. Those constraints are used to select test formulae based on its global shape, or on its events. Elementary constraints can be combined to form complex constraints. We have also shown how the variables not fixed by these constraints can be instantiated using uniformity hypotheses, and how to enhance the quality of the selection with subdomain decomposition. Although our method solves the most common problems in testing O-O software, it does not deal with polymorphism and inheritance. These issues are examined in [Barbey, 1997].

Our approach has similarity with the approaches presented in section 2. Like Kirani, we have a model for possible sequences of messages. However, unlike Kirani, we use the axioms of the specifications to partition the parameter values, like ASTOOT. Unlike ASTOOT, we do not limit ourselves to term rewriting, but generate longer combinations of invocations corresponding to repetitions and interleaving of events. We take better advantage of the specification,e.g. by using axioms for classifying methods (state of interest, category). Also, our approach is different in that it does not specify what to test, but what not to test (i.e. which test cases to remove from the exhaustive test set) based on the hypotheses. Thus, the quality of the test set only depends on the quality of the hypotheses. However, in the present state of the art, how to measure the quality of hypotheses is still an open issue.

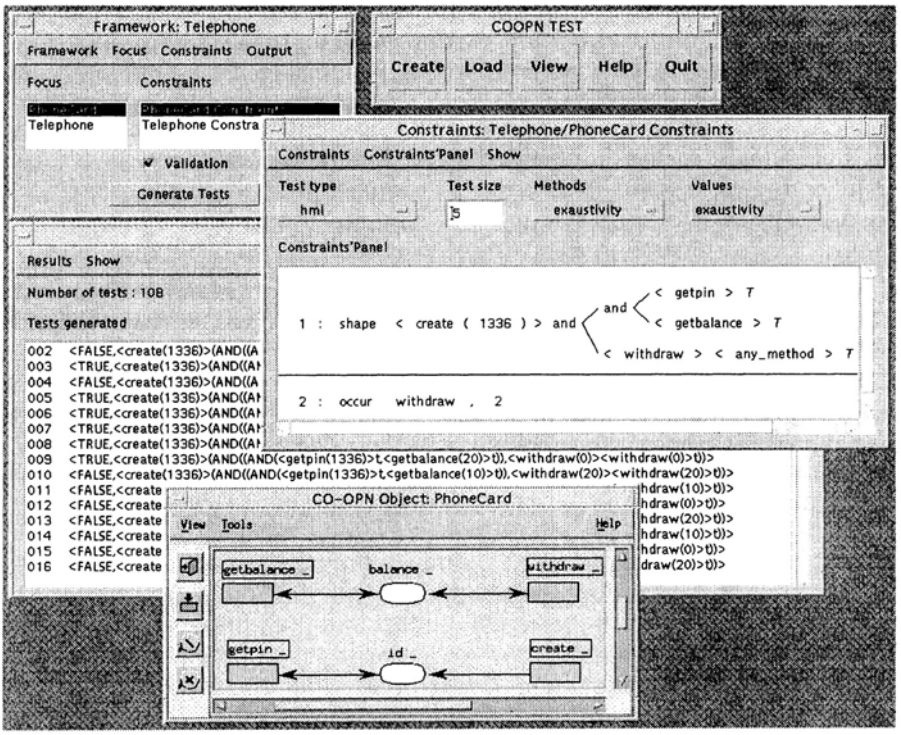

Figure 4. Snapshot of the test selection for the class PhoneCard with *CO-OPNTest*

Our theory and method also exhibit the advantage of being formal enough to semi-automate the test selection. A Prolog tool, called *CO-OPNTest*, is being implemented, by coding the *HML* semantics rules and the *CO-OPN/2* class semantics into equational logic: resolution techniques allow us to compute test sets from a *CO-OPN/2* system and the constraints presented in this paper. A front-end, written in Java, allows a user-friendly definition of the constraints. The figure 4 displays the snapshot of the test of a phone card. The 108 test cases generated are constrained by the constraints *shape* and *occur* (*nb-occurrences* in the text), and by a size (*nb-events*) of 5. They have been generated in a few seconds. Moreover, the tool allows a graphical representation of the Petri nets of the objects in the system. This representation is useful because it allows an intuitive comprehension of the specification and thus guides the tester during the test selection process.

Although *CO-OPNTest* is still under development, it already allowed us to generate test sets for several case studies in a simple, rapid and efficient way. Moreover, it shows how to apply our theory and demonstrates the pertinence of our approach.

8 ACKNOWLEDGMENTS

We wish to thank Bruno Marre, Pascale Thévenod, Hélène Waeselynck, and the anonymous referees for their numerous and valuable comments on previous versions of this paper.

This work has been supported partially by the Esprit Long Term Research Project 20072 "Design for Validation" (DeVa) with the financial support of the OFES, and by the FNRS project 20-43648.95 "Research in O-O Software Development".

9 REFERENCES

Arnould, A. (1997). *Test à partir de spécifications de structures bornées: une théorie du test, une méthode de sélection, un outil d'assistance à la sélection.* PhD thesis, Université de Paris-Sud, U.F.R. scientifique d'Orsay.

Barbey, S. (1997). *Test Selection for Specification-Based Testing of Object-Oriented Software Based on Formal Specifications.* PhD thesis, Swiss Federal Institute of Technology in Lausanne (EPFL). Ph.D. Thesis 1753.

Barbey, S., Buchs, D., and Péraire, C. (1996). A theory of specification-based testing for object-oriented software. In *Proceedings of EDCC2 (European Dependable Computing Conference)*, LNCS (Lecture Notes in Computer Science) 1150, pages 303–320, Taormina, Italy. Springer verlag.

Bernot, G., Gaudel, M.-C., and Marre, B. (1991). Software testing based on formal specifications: a theory and a tool. *IEE Software Engineering Journal*, 6(6):387–405.

Biberstein, O., Buchs, D., and Guelfi, N. (1997). COOPN/2: A concurrent object-oriented formalism. In Bowman, H. and Derrick, J., editors, *Proc. of FMOODS '97*, pages 57–72, Canterbury, UK. Chapman and Hall, London.

Binder, R. V. (1996). Testing object-oriented software: a survey. *Journal of Testing, Verification and Reliability*, 6:125–252.

de Nicola, R. and Hennessy, M. (1984). Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133. North-Holland (Elsevier).

Doong, R.-K. and Frankl, P. G. (1994). The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130.

Hennessy, M. and Milner, R. (1985). Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161.

Hennessy, M. and Stirling, C. (1985). The power of the future perfect in program logics. *Information and Control*, 67(1-3):23–52.

Kirani, S. (1994). *Specification and Verification of Object-Oriented Programs.* PhD thesis, University of Minnesota.

10 BIOGRAPHY

Cécile Péraire is a research assistant at the Software Engineering Laboratory of EPFL. She graduated from EPFL in 1993. She is a doctoral candidate.

Stéphane Barbey is a research assistant at the Software Engineering Laboratory of EPFL. He graduated from EPFL in 1992 and holds a Ph.D. since 1997.

Didier Buchs is a senior researcher at the Software Engineering Laboratory of EPFL since 1993. He obtained a Ph.D. degree in Computer Science from the University of Geneva in 1989.