Cover Page

Title

Incremental Test Selection for Specification-Based Unit Testing of Object-Oriented Software Based on Formal Specifications

Authors

Stéphane Barbey, Didier Buchs, Cécile Péraire, and Alfred Strohmeier

Affiliation

Swiss Federal Institute of Technology
Software Engineering Laboratory

Abstract

We propose a test case selection methodology for object-oriented software. This methodology is based on formal specifications and is aimed at verifying the correctness of method interaction of an object or a cluster of objects. The essence of this methodology is to select test cases by a reduction process. The exhaustive test set, the reference for correctness, is transformed into a practicable test set by applying hypotheses to the objects. We present a set of hypotheses specific to object-oriented systems. Regularity hypotheses are m:n generalizations of the unit behaviour and provides for specifying the shape of test cases. Uniformity hypotheses are 1:n generalizations. Used in conjunction with subdomain decomposition, they guarantee a coverage of the specification. Incrementallity hypotheses take advantage of the subtyping and subclassing relationships to justify the reuse of test cases.

Kind

Research paper

Subject area

object testing; theoretical foundations; programming environments

Contact information

Swiss Federal Institute of Technology
Software Engineering Laboratory
1015 Lausanne, Switzerland
phone: +41 (21) 693.52.43, fax: +41 (21) 693.50.79

email: {Stephane.Barbey, Cecile.Peraire, Didier.Buchs, Alfred.Strohmeier} @ epfl.ch

url: http://lglwww.epfl.ch

Incremental Test Selection for Specification-Based Unit Testing of Object-Oriented Software Based on Formal Specifications

Stéphane Barbey, Didier Buchs, Cécile Péraire, and Alfred Strohmeier

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

ABSTRACT. *We propose a test case selection methodology for object-oriented software. This methodology is based on formal specifications and is aimed at verifying the correctness of method interaction of an object or a cluster of objects. The essence of this methodology is to select test cases by a reduction process. The exhaustive test set, the reference for correctness, is transformed into a practicable test set by applying hypotheses to the objects. We present a set of hypotheses specific to object-oriented systems. Regularity hypotheses are m:n generalizations of the unit behaviour and provides for specifying the shape of test cases. Uniformity hypotheses are 1:n generalizations. Used in conjunction with subdomain decomposition, they guarantee a coverage of the specification. Incrementallity hypotheses take advantage of the subtyping and subclassing relationships to justify the reuse of test cases.*

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

1. Introduction

Although the use of object-oriented principles increases the quality of software by leading to better, modular architectures, it does not provide correctness by itself, because "*software is created by error-prone human*" (Boehm). Since the object-oriented technology promotes reuse, it could be argued that the need for meticulously tested components is even more important. Thus, testing remains an important task even for object-oriented software development.

Although testing object-oriented software has been recognized as an important subject of research, all dimensions of object-oriented programming relevant to testing have not been covered. Previous work has focused on testing class-based systems ([8], [14], [15]), and to some extent, inheritance of implementation ([18], [11]). In this paper, we propose a test selection method for unit testing of object-oriented software [3]. The main difficulties in selecting test cases for object-oriented software unit testing are that methods cannot be tested individually, but must be tested in interaction, and that, because of encapsulation and hiding, the oracle — the process that validates the result of a test— must be based on an external observation of the objects behavior.

The theoretical framework of our approach is an adaptation to O-O software of the theory of testing by Bernot, Gaudel and Marre [5]. Its essence is not to select test cases using coverage criteria as measure of quality, but to start from the exhaustive test set which covers all possible interactions of the methods of an object or of a group of objects and serves as reference of correctness, and to reduce it into a finite and pertinent test set by reduction hypotheses. Thus, the quality of the test set is a measure of the quality of the hypotheses.

In section 2, we present the theoretical grounds of this paper. This section includes an introduction to testing, and we describe a specification language well-adapted to our requirements, *CO-OPN/2*, and a format for test cases, formulae based on the Hennessy-Milner temporal logic, that is well-adapted to capture methods interactions and to build simple oracles.

In section 3, we present a practical process for test selection based on the application of hypotheses. In section 4, we define several regularity hypotheses, which are *m:n* generalizations of the unit behavior. Those hypotheses are used to select test cases based on a shape, on the state of the involved objects, or on their events. In section 5, we show how to instantiate using uniformity hypotheses, which are *1:n* generalizations of the unit behavior, and how the quality of test sets can be enhanced by combining uniformity hypotheses and subdomain decomposition.

In section 6, we propose hypotheses of incrementallity to take subclassing and subtyping into account. Capitalizing on these relationships can be very useful in minimizing the effort of testing a descendant class given the test set developed for the parent class.

In section , we conclude this paper by briefly presenting *CO-OPNTest*, a tool that semi-automatizes the selection process.

2. Theoretical Grounds

This section defines some terminology related to testing and provides a short overview of the theory behind testing. The interested reader may refer to [20] for a more comprehensive introduction.

Testing is the most popular verification technique. For unit testing, it aims at answering the following question: “Does a unit U correctly implement a model M?” In order to do so, it tries to find counterexamples showing that it does not [16]. This verification requirement can be summarized by the following equation:

$$(U \models M) \Leftrightarrow (U \models_O T)$$

The test process involves three steps. It is decomposed in:

- A test selection step, in which the tester must select a test set T such that if the unit satisfies T, then it also satisfies its model M. A *test set* is a set of test cases. A *test case* is an input on which the unit under test is executed during testing. The test selection procedure should also decide what the expected results of the test sets are.
- A test execution step, in which the test cases are executed.
- A test satisfaction step, in which the results obtained during the test execution phase are compared to the expected results using the oracle $\models$. The *oracle* is the process, be it a program or human inspection, that compares the result generated with the one expected by the model.

The results yielded by this comparison give a test result for the verification requirement. Possible results include yes if it is validated, no it is not, and undecided if the oracle was unable to analyze the result or if the test execution did not return a result.

The answer to the verification requirement is undecidable in the general case because only the exhaustive test set satisfies the previous equation. The oracle $\models_O$ can only be constructed for a unit U and a test set T if T is practicable, i.e.: T is tractable, i.e. it has a "reasonable" finite size, which is not the case of the exhaustive test set for arbitrary specifications, and T is decidable, i.e. it is possible to decide whether the unit is acceptable or not for the given specification. So, in order to have a practicable test set, a trade-off must be found between size and accuracy. Already in the mid-70s, Goodenough and Gerhart [10] pointed out that the central question was therefore to find a *test criterion*, i.e. a criterion that states what is an adequate test set. Thus, the verification requirement becomes:

"How well does a unit U implement a model M according to a sampling criterion C?"

A great number of such criteria have been proposed and investigated. The criterion is often defined by *sampling* the input domain of the unit to exercise it with interesting cases only.

Among the several test selection techniques, specification-based test selection is an approach that only uses the specification of the model, and does not use the tested unit itself. Specification-based 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 (see section 2.3).

In the remaining of this section, we will first present the hypotheses approach to test quality, and then a specification language, *CO-OPN/2*, and a language for expressing test cases, *HML*.

2.1 Quality of Testing and Hypotheses

The quality of a test set is measured by its *adequacy*, which needs appropriate measurement techniques. Adequacy is usually a function of the coverage of the sampling criteria by a test set. For s*pecification-based testing* adequacy might be measured by coverage of axioms as found in the formal specification, or coverage of the input domain after partitioning in equivalence classes. However, for object-oriented software, finding adequate coverage criteria is difficult.

- The coverage of a method is not only defined by the values that its parameters can take, but also by the state of the object on which the method is applied. Methods cannot be invoked alone, they have to be applied on an object, and their behavior does not only depend on the input domain, but also on their history.
- The state of the object is itself difficult to determine, because it does not only include the attributes of the object, but also the references to other objects. A method can have no effect on the state of the object on which it is applied, but produce modifications on the state of connected objects.

So covering methods individually is not sufficient for covering a class. Methods should not be considered as basic test units, and they should not be tested as if they were free floating operations, but *only in interaction*. However, since the number of interactions of the number of a class is generally not bound, a coverage criteria for these interactions cannot be easily expressed.

An alternative to coverage criteria is to apply selection hypotheses to the unit. Hypotheses state under which conditions the satisfaction of the specification is ensured by the satisfaction of the test set by making assumptions that the unit reacts in the same way for some test data: hypotheses correspond to *generalizations* of the behavior of the unit. The idea behind hypotheses is to explicitly state what cases sampling will exclude, i.e. hypotheses are applied to a unit to enumerate the cases that the tester will not test, because its makes the hypotheses that the unit behaves correctly for those cases given testing performed for other cases. Therefore, instead of specifying what to test, the tester specifies what it will not test.

Assuming that hypotheses H have been made on U, the previous equation becomes:

$$(U \text{ satisfies } H) \Rightarrow (U \models_O T \Leftrightarrow U \models SP).$$

A nice property of this equation is that the quality of the test set T is only dependent on the quality of the hypotheses. The drawback however is that proving that a unit satisfies the hypotheses is not trivial.

Thus, test selection consists in selecting from the exhaustive test set a finite set of ground test cases which is sufficient to prove the property preservation of a unit with respect to its specification. Validity and unbias are necessary properties of the reduced test sets, i.e. after the application of the hypothesis, all unsatisfied tests should reveal program errors and no satisfied test should hide errors. In our case, reduction of the exhaustive test set to the final test set, i.e. the ground test cases, is performed by applying hypotheses to the unit (see figure 1).

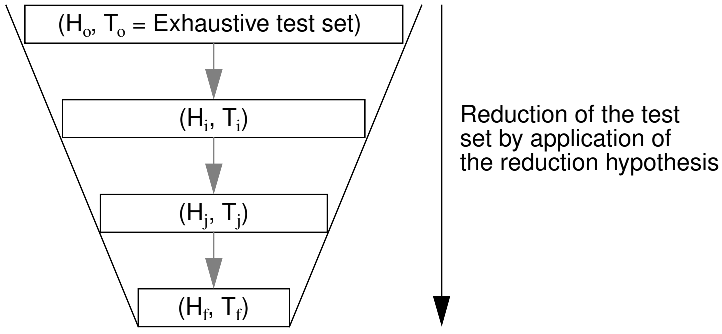

Fig. 1. Iterative refinement of the test context

We can define test selection as the iterative refinement of a context defined as a couple (H, T) where H is the set of hypotheses, and T is the test set. The selection process starts from a context (H_o, T_o) which corresponds to the exhaustive test set and no reduction hypothesis. The selection of a practicable test set *T* is made by successive refinements of an initial test context (H_0, T_0) which has a pertinent test set T_0 (possibly of not "reasonable" size), until obtaining a practicable test context (*H*, *T*): $(H_0, T_0) \leq ... (H_i, T_i) \leq (H_j, T_j) ... \leq (H, T)_O$. The introduction of a new hypothesis *h* provides a new testing context (H_j, T_j), where $H_j = H_i \cup h$, and T_j is a subset of T_i ($T_j \subseteq T_i$). The context refinement is performed iteratively until we reach a test set (H_f,T_f) that is finite, decidable, and preserves the properties of the exhaustive test set.

2.2 Specifying Object Systems

The work presented in this document does not require the use of a specific specification formalism. Meanwhile, to achieve our goals, it is necessary to rely on a specification of the unit under test that is as complete and consistent as possible. Although they cannot ensure entire completeness and consistency — which will remain the task of the programmer —, the use of a formal specification language can only be beneficial for our purpose. Therefore, we have settled to use the formalism *CO-OPN/2* (Concurrent Object-Oriented Petri Nets/2) for specifying our test units. *CO-OPN/2* ([7], [6]) belongs to the category of high-level Petri nets, which are now widely used in both theoretical analysis and practical development of concurrent systems. *CO-OPN/2* is designed for the architecture and the specification of large and complex concurrent systems. It provides a sound mathematical formalism, allowing the expression of the particularities of complex systems through the use of two underlying formalisms, algebraic specifications, to describe data structures and functional aspects of a system, and Petri nets, to model its concurrent aspects.

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).

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.

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). The

```
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 2. Textual specification of the classes Phonecard and Telephone

figure 2 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.

2.3 Specifying Test Cases

The test cases can be expressed with any logic compatible with $\models$. For *CO-OPN/2*, the semantics of which is expressed with transition systems, and which allows the comparison of two specifications with the strong bisimulation principle, the *HML* logic introduced by Hennessy-Milner in [12] satisfies this requirement [4]. *HML* is a language centered on the notion of events (or messages in an object-oriented friendly terminology). It is very simple, in that it only includes four operators: *Next* (<_>), *And* ($\wedge$), *Not* ($\neg$), *T* (always true constant), The language based on the events of the specification *SP* and the *HML* operators will be called HML_{SP}. The advantage of using *HML* as for expressing test cases are its simplicity, which will allow the construction of simple oracles, because *HML* provides an observational description of the valid implementation through the test cases, and its theoretical foundations: there is a full agreement between *HML* and *CO-OPN/2*.

An elementary test case for a unit under test *U* and a specification *SP* 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*.

A test case <*Formula*, *Result*> is successful if *Result* reflects the validity of *Formula* in the labeled transition system modeling the expected behavior of *U*, 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 unit (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*, which includes all possible behaviors derivable from the specification, as:

$$\textit{EXHAUST} = \{\langle f, r\rangle \in HML_{SP} \times \{true, false\} \mid (SP \models f \text{ and } r = true) \text{ or } (SP \not\models f \text{ and } r = false)\}.$$

3. The Test Selection Process

The practical test selection process is actually implemented as a selection process that emulates the theoretical reduction process [17]. The idea is to consider the exhaustive test set as a formula *f* which is a formula with variables universally quantified, i.e. *f* is a variable that stands for all its possible instantiations. The aim of test selection becomes the reduction of the level of abstraction of the formula *f* until it is ground. We propose the four following steps to implement the practical test selection process: *i)* focus on a unit of test, *ii)* define the test environment, ii*i)* define selection hypotheses, *iv)* select test cases.

3.1 Focus on a Unit of Test

There are two trends on how method interaction testing should be performed: the structural trend ([9], [2]), and the behavioral trend ([13]). The structural trend is focused on intra-class method interactions. It considers classes as test units, and aims at testing the interactions of the methods of a single object. The levels of granularity coincide with the scope of program units, from the compiler point of view. In this approach, the basic units of test are the classes, which are integrated into systems, or possibly into subsystems. The behavioral trend is focused on object communication. Jorgensen and Erickson [13] dismiss the importance of object and class integration, asserting that software testing is primarily concerned with what the software does—its behavior— and not how it is organized —its structure. The behavioral trend considers the meaningful subsequences of system operations as test units, and verifies how the methods of different objects interact to provide a system operation.

Those two trends however are not irreconcilable and none of them is really satisfactory.

- Proponents of the behavioral trend assume that system operations should be tested individually. However, for a system operation to be invoked on a system, it is necessary to put the system in an initial state, in which the preconditions of the system operations are satisfied by the system. This state can only be reached by involving methods of those objects, or by providing a large number of stubs.
- Object-oriented software development is concerned by reuse, which occurs at a static level. Classes are the basic units of reuse, and at a higher level, subsystems are also reusable, whereas neither methods nor system operations are reusable. Therefore, it is important to obtain confidence in classes.
- Objects sometimes mutually depend on each other, so that an object cannot be tested alone through its interface. In that case, it may be better to give up individual class testing in favor of cluster testing [19].
- There is more in classes than just objects. Classes also include a behavior and a state of their own (object management utilities such as creation/destruction methods, class variables, class methods,...). Thus, it is important to test the object management part at one level, and the class management part at another level. Testing the class management part of a class requires instantiating several objects, for instance, to test that a different object identity is assigned to objects, or that there are no errors in handling the state shared among instances of a class (the class variables).

Therefore we postulate that, since methods are not efficient basic test units, and the structure of object-oriented systems does not allow a recursive decomposition that takes all behavioral aspects into account, there is no *basic unit testing* for object-oriented software, but there are many kinds of *integration unit testing*:

- *Object integration* tests the cooperation of the methods of the object management part of one class with one object of this class. This kind of testing checks that an object satisfies its specification (or, of course, to find counterexamples where it does not) for the object management part. Object integration testing may of course include creators, which are class methods.
- *Class integration* tests the cooperation of the methods of the class management part of one class with several objects of the same class. The methods of a class are tested in integration to check the class management part of a class. Models from which test data can be derived for object integration and class integration include class formal specifications,...
- *Cluster integration* tests the collaboration of all methods of several classes with instances of these classes. It is required when objects are too closely tied to be tested individually. Models from which test data can be derived for cluster integration are the same as for object and class integration, but cluster integration must consider the union of these models.

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.

So the first step in our test selection process will be to focus on a particular unit of interest, the focus, that we want to test in detail. This unit can be an object (object integration testing), a class (class integration testing), or even a cluster (cluster integration testing). For instance in figure 3, the focus is A, which uses the units B and C. The unit A can be tested using already tested implementations of B and C or stubs that simulate the behavior of B and C.

3.2 Deduce the Test Environment from the Focus

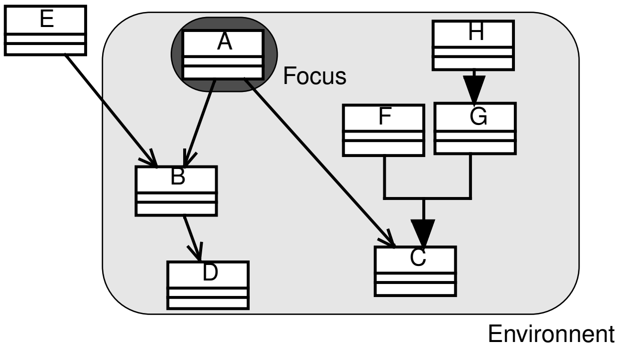

Fig. 3. Focus and environment

The test environment is the set of all the units visibly (i.e. appearing in the interface) used by the focus of interest. This test environment includes all units that are directly and indirectly used. For instance, in figure 3, the class B uses the class D, and that class must be included in the environment if it appears in the interface of B, because it may be useful to create or modify the behavior of B. The test environment must also include the subtypes of used units, because of the possibility of substitutions. For instance, although the classes D, F, and G may not be imported in the specification of A, they must be integrated in the test environment because C is imported, and objects of the classes F, G, and H can be substituted for objects of the class C.

For object integration testing, the unit under test will be tested through one of its instances, the object under test. For class integration testing, several objects under test of the same class will be selected. Since *CO-OPN/2* has neither a metaclass level, nor class attributes, several instances of the class will only appear if needed in methods parameters. For cluster integration testing, several objects under test belonging to the classes in the cluster will be selected.

3.3 Define Selection Hypotheses

This is achieved by defining a system of hypotheses on the non-ground exhaustive test set with the help of selection hypotheses The non-ground exhaustive test set is described by $\langle f, Result \rangle$

We present three hypotheses to limit the size of the test set, by reducing the number of cases that must be tested, while preserving the integrity of the test set. These hypotheses are the uniformity hypotheses, which are strong and correspond to a 1:*n* generalization of the unit behavior, the regularity hypotheses, which are weaker and correspond to a *n*:*m* generalization of the unit behavior, and the incrementallity hypotheses, which take into account the property preservation of subtyping and subclassing.

Since the weights of those hypotheses are unequal, they are not to be applied for the same purpose (see figure 4).

- For the focus of interest: use of *weak* reduction hypotheses (like regularity) so that assumptions made on non tested units are minimal.
- For the environment: use of *strong* reduction hypotheses (like uniformity) 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 by unfolding techniques.

The incrementallity hypotheses are particular in that they do not apply for the construction of individual test cases, but for the construction of test sets based on other test sets.

3.4 Solve the System of Hypotheses

This practical test selection process results in a practicable test set. From that point, the test cases are solved by computation of the value of the variable *Result*: *true* if the HML_{SP} formula describes an acceptable behavior of the unit, and *false* otherwise.

Of course, to select a complete test set for a given class, it may be necessary to define several systems of hypotheses 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 formula, Result*>, and because of its construction's

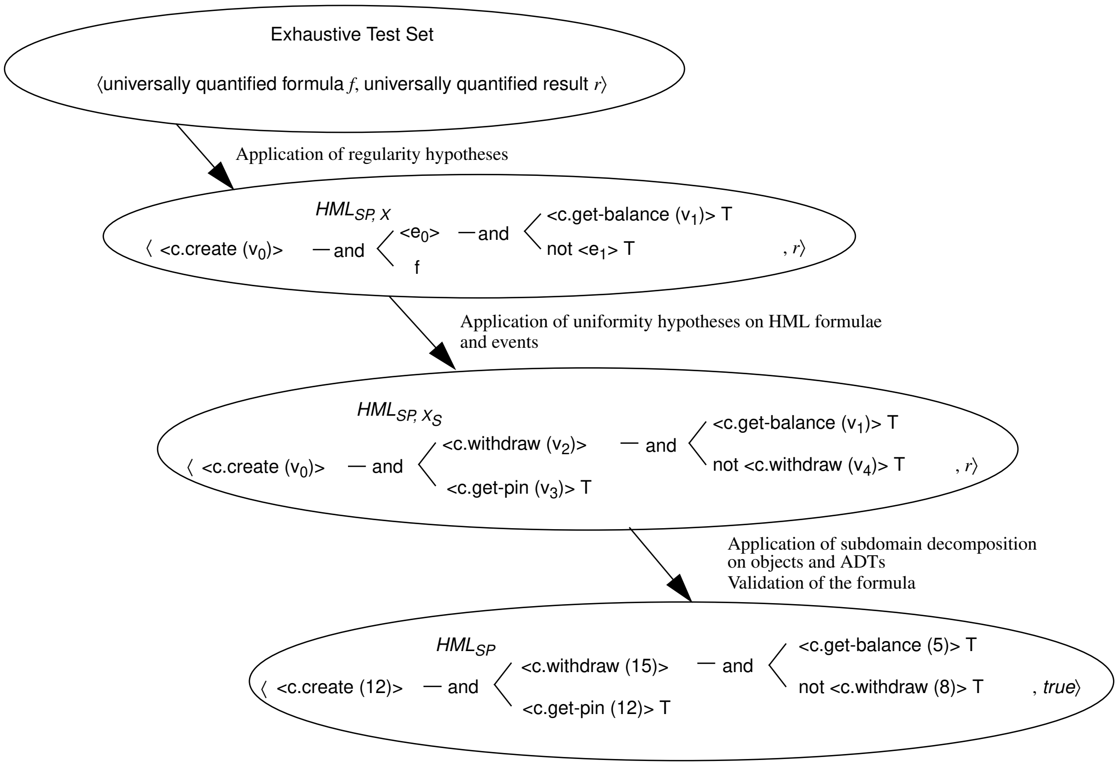

Fig. 4. Application of hypotheses

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 validity and unbias. 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 redundancies, a test set can be transformed into a test set free of redundant test cases at any time during the process.

4. Regularity Hypotheses

It is often reasonable to start the test set selection process by applying regularity hypotheses to the unit. The regularity hypotheses allows reducing the level of abstraction of the initial tests cases, the unground formula *f*, by giving it a shape. Then uniformity hypotheses can be applied, i.e. the instantiation of the remaining variables of the formulae can be constrained. This section presents different regularity hypotheses. As mentioned in [1], the strategies need to be carefully built to implement the corresponding hypotheses. The danger is to obtain test sets which do not keep the good properties of the initial exhaustive test set: validity and unbias.

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 hypothesis *C*, then it is successful for all possible instances of *v*. For instance, in the case of the telephone system and a hypothesis *“the number of insertions of a phonecard is equal to 20“*, if all test cases $<f, r>$, in which the hypothesis is satisfied, are successful, then the system behaves correctly for all possible numbers of insertions.

More specifically, we have defined three kinds of regularity hypotheses.

4.1 Regularity on Events

Regularity hypotheses on events can be stated to limit the size of formulae, independently of their contents. Examples of regularity on events include the following hypotheses:

- *Hypothesis nb-events (f):*

If a test case $\langle f, r \rangle$ 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.

- *Hypothesis depth (f):*

If a test case $\langle f, r \rangle$ is successful for all instances of f having a depth equal to a bound k, then it is successful for all possible instances of f. (The depth of a formula corresponds to the length of its paths.)

- *Hypothesis nb-occurrences (f, m)*

If a test case $\langle f, r \rangle$ 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.

For both the *nb-events* and the *depth* hypotheses, it is very difficult to prove that the unit satisfies the associated hypothesis. Thus, those hypotheses should only be used as a barrier to avoid a combinatorial explosion, and the other (semantics-oriented) hypotheses should be used to select meaningful test cases. In practice, it is better to use the *nb-events* and *depth* hypotheses to match all formulae lower or equal in size, respectively in depth, than to choose an exact size or depth, i.e. to select the predicates $nb\text{-}events\ (f) \leq k$ or $depth\ (f, l) = true,\ l \leq k$.

4.2 State and Method Classification

State and method classification techniques can be used as regularity hypotheses to take into account the semantics of the individual methods.

- *Hypotheses only (f, i) (i=constructor, mutator, observer)*

The events used in the test cases are based on the kinds of the methods of the environment. The kinds of actions performed by the events are classified into *constructor*, *mutator* and *observer*. For instance, in the class PhoneCard of the telephone example, the events based on create are constructors, the events based on withdraw are mutators and the events based on get-pin and get-balance are observers.

So the hypothesis will be that if a test case $\langle f, r \rangle$ is successful for all instances of f which are a combination of constructors followed by a combination of mutators and terminated by a combination of observers, then it is successful for all possible instances of f.

- *Hypothesis only (f, $\mathcal{P}(m)$)*

It is common that the methods of a class do not all act on parts of the state of its instances, but only on some of them, and that 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, testing those interactions will not help much in discovering errors. This division can be based on the pre- and postconditions of the axioms, which define how events read and modify the contents of places.

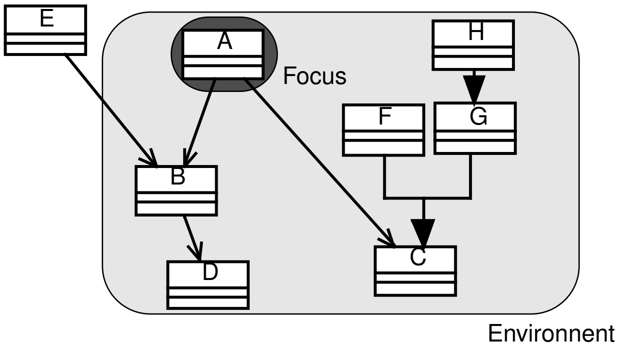

Fig. 5. Focus and environment

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, whereas the methods get-balance and withdraw are involved in the accounting of the card, and are connected to the place balance (see figure 5).

So the hypothesis will be that 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 set of test cases for $M\ (P_i)$ and another set of test cases for $M\ (P_j)$ are successful, all test cases for $M\ (P_i \cup P_j)$ are successful too.

4.3 Regularity on the Shape of Formulae

This section presents a hypothesis applicable on the shape of the $HML_{SP,\ X}$ formulae, i.e. a hypothesis allowing to force the shape of test cases. In that case, the hypothesis is that if a test case $\langle f, r \rangle$ is successful for all instances of f having a given shape s, then it is successful for all possible instances of f. For instance, the hypothesis

C: shape (f, <c.create (1234)> (f_0 and not <c.get-balance (25)> f1)) = true

leads to the test case

⟨<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* formulae. The hypothesis C could be applied in conjunction with other hypotheses. For instance the conjunction

C $\wedge$ (*nb-occurrences* (*f_0*, *get-pin) = 1*) $\wedge$ (*shape* (*f_1*, *T) = true*)

is a hypothesis leading to this kind of test cases

⟨<c.create (1234)> (<c.get-pin (v_0)> *T* and (not <c.get-balance (25)> *T)),* result ⟩

⟨<c.create (1234)> (<c.get-pin (v_0)> <get-balance (v_1)> *T* and (not <c.get-balance (25)> *T*)), result ⟩

in which the variable f_0 has been instantiated with *HML* formulae obtained by application of the hypothesis *nb-occurrences (*f_0, get-pin*) = 1* and the variable f_1 has been instantiated by *HML* formulae obtained by application of the hypothesis *shape (*f_1*, T) = true*. Defining the shape of test formulae as the concatenation of several shapes also has the advantage of allowing setting the state from which the test selection must occur. For instance, to test the behavior of an empty phonecard, it is possible to first specify the shape

C1: shape (g, <c.create (v_0)> <c.witdraw (20)> f_0) = true

and then to select test cases with, for instance, the hypotheses

C2: (f = g | h) and shape (g, <c.create (v_0)> <c.withdraw (20)> f_0) = true and nb-events (g) = 4 and nb-occurrences (g, withdraw) = 2

The generated test sets will test the interaction of the methods of PhoneCard, with regard to further hypotheses that can be defined on *g*, on an empty phonecard.

5. Uniformity Hypotheses and Subdomain Decomposition

The application of regularity hypotheses on the exhaustive test set generates test cases with remaining variables. These variables can be replaced using various strategies, like *exhaustiveness* 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 remaining variable. For instance, for example of the telephone system, a uniformity hypothesis can be applied on the pin-code passed to get-pin, because the specification of PhoneCard does not depend on its value (no properties of the pin-code are used, but for the equality), but on its existence. The strategy for uniformity of the four kinds of variables that can occur in a formula *f* is the following:

- uniformity on *HML* formulae: Any *HML* formula can be randomly selected, with respect to the hypotheses applied on the enclosing test formula.
- uniformity on events: Any event can be randomly selected, with respect to the hypotheses that are 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 unit can show the validity of a uniformity hypothesis by examining the use of the object on which the uniformity hypotheses are applied.

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, it consists in finding the possible behaviors encompassed in *f*, and, for each found behavior, to select a test case in which the remaining variables are assigned values that will cause this behavior. In *CO-OPN/2*, variables with a domain belong to algebraic values (ADTs) and objects (classes).

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. The constraints drive the possible executions. They can influence the ability to trigger an event, and are algebraic or synchronization conditions. 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). The 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 constraints are drawn up, in concordance with the results of the condition. We include a constraint for which the condition yields true and another one for which it yields false.
- 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.

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.

For instance, given the following formula, which includes an *HML And* operator, and satisfies the hypothesis ($f = ((f_c \mid f_m) \mid f_o) \wedge only_{creator}(f_c) \wedge only_{modifier}(f_m) \wedge only_{observer}(f_o)) = true$):

f = <c.create V_0> <c.withdraw V_1>(<c.get-pin V_2> and <c.get-balance V_3>)T

The only condition is on the method withdraw, from which we extract two constraints corresponding to the case of the balance bigger than the withdrawn money, and the opposite case. From the *CO-OPN/2* derivation trees (which are not shown), we can extract the set of subdomains *S*.

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

The constraints β correspond to the constraints found out in the axioms, and the constraints σ to the constraints that make up the glue between the events. 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 — Computation of subdomains from the constraints: Unfold the constraints to find uniformity subdomains on algebraic axioms.

In our example, 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, $(20–V_1 = V_3)$ = true}, f, true>

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

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

4: <{V_0, V_1, V_2, V_3}, {$(20 \geq V_1)$ = true, $(V_0 = V_2)$ = false, $(20–V_1 = V_3)$ = 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 — Test set selection from the subdomains: For each uniformity subdomain, solve the constraints and select values for the variables.

In our example, 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>

6. Incrementallity Hypotheses

In this section, we study techniques of incremental testing that capitalize on subtyping and subclassing. When selecting test sets, we can share test information between groups of related classes: classes are built on top of each other, and can thus be tested in a joint fashion, by testing completely the root of the hierarchy and then only testing increments for descendant classes. *CO-OPN/2* provides two means of structuring specifications into hierarchies: subclassing and subtyping. Subclassing is a relationship between modules, whereas subtyping is a relationship among types. Inheritance is the syntactic mechanism to implement subclassing. It allows reuse and refinement of a class, whereas subtyping address a semantic conformance relationship between types. Inheritance allows a selective selection of inheritance components, permitting to rename, redefine and undefine components, whereas subtyping limit the enrichment to what is allowed by the strong substitution principle.

The incrementallity hypotheses take advantage of the complex, but strong relationship between the test sets of two specifications for two units, so that unchanged methods need no further testing, and so that test sets defined for a parent class to reduce the test sets to be selected to test the descendant class. The verification requirement becomes:

"How well does a unit U_2 implement a model M_2 given that the unit U_1 implements the model M_1 according to the criterion C?"

To discern which test cases must be reused and which must be dropped, we must discern several cases:

- test cases that are dropped because either they do not apply for the descendant, or because, although they apply, they need not be run because there were no modification from the parent to the descendant,
- test cases that are reused, they still apply, and must be run.

More than a hypothesis, an incrementallity hypothesis is an hypotheses about hypotheses: it states that if the hypotheses stated for testing U_1 were right, then they are also right for the part of U_2 common to U_1. Thus, the problem of test selection becomes finding the set of properties that must be tested for the unit U_2 with regard to the hypotheses stated on the unit U_1. This is accomplished with a function *Diff* that calculates the difference between two classes.

Thus, the whole problem is to define a correct function *Diff*. The effects of inheritance are not limited to added and redefined methods, but include some of the inherited methods, because of history errors and inter-class dependencies among methods. To calculate *Diff*, we discern three orthogonal areas of inheritance between two classes: inheritance of interface, of semantics, and of implementation:

- Inheritance of interface: the inherited properties are the signatures of methods: their name, their formal parameters, and, in the case of functions, the type of their result.
- Inheritance of semantics: the inherited properties are the descriptions of the intended behavior and are found in the specification's body.
- Inheritance of implementation: the inherited properties are the code of the program.

For each area, four cases can occur:

- Properties are undefined, i.e. they are removed from the parent and do not appear in the descendant.
- Properties are redefined, i.e. they are removed from the parent, and a new definition is given to them in the descendant.
- Properties are added, i.e. they are not present in the parent, but only in the descendant.
- Properties are unchanged, i.e. they are present in the parent and in the descendant.

The complete algorithm for calculating the function *Diff* can be found in [3] and is summarized for both subclassing and subtyping in the table below (R, M, K stand respectively for Removed, Modified and Kept). This table only shows the individual elements, and does not take into account their conjunction.

<table>
<tr><th colspan="5" rowspan="2"></th><th colspan="3">Subclassing</th><th colspan="3">Subtyping</th></tr>
<tr><th colspan="3">Implementation</th><th colspan="3">Implementation</th></tr>
<tr><th colspan="5"></th><th>R</th><th>M</th><th>K</th><th>R</th><th>M</th><th>K</th></tr>
<tr><th rowspan="10">Semantics</th><th rowspan="4">K</th><th rowspan="4">Interface</th><th colspan="2">K</th><td rowspan="2">reused (for a possible descendant)</td><td rowspan="2">reused</td><td rowspan="2">dropped (although applicable)</td><td>reused (for a possible descendant)</td><td>reused</td><td>dropped (although applicable)</td></tr>
<tr><th rowspan="2">M</th><th>covariance</th><td>reused (for a possible descendant), new test cases must be selected</td><td>reused, new test cases must be selected</td><td>dropped (although applicable), new test cases must be selected</td></tr>
<tr><th>contravariance</th><td>reused (for a possible descendant), new test cases must be selected</td><td>reused, new test cases must be selected</td><td>dropped (although applicable), new test cases must be selected</td><td>reused (for a possible descendant)</td><td>reused</td><td>dropped (although applicable)</td></tr>
<tr><th colspan="2">R</th><td colspan="3" rowspan="7">dropped (non applicable), new test cases must be selected</td><td colspan="3" rowspan="7">(not applicable to subtyping)</td></tr>
<tr><th rowspan="3">M</th><th rowspan="3">Interface</th><th colspan="2">K</th></tr>
<tr><th colspan="2">M</th></tr>
<tr><th colspan="2">R</th></tr>
<tr><th rowspan="3">R</th><th rowspan="3">Interface</th><th colspan="2">K</th></tr>
<tr><th colspan="2">M</th></tr>
<tr><th colspan="2">R</th></tr>
</table>

New test cases must of course be selected for the added methods. For redefined methods, new test cases may possibly be added to take into account the new properties of the methods. Test cases containing unchanged methods do not automatically escape to retesting, because their behavior may be modified by changes in the behavior of other methods. Finally, methods which do not appear in the above cases do not need retesting: the test run on the parent type are sufficient to show that those methods meet their specification.

Determining whether a test case is reused or dropped is performed by taking in conjunction the status of the different methods that compose the test case, according to the function *Diff*. Once again, the complete algorithm can be found in [3]. However, these rules obey to the following order of precedence: i) the test case must be dropped, ii) the test case must be reused, iii) the test case can be dropped or the test case can be reused.

If the above rules imply that a test case must be dropped and must be kept, it will be dropped from the test set. If they imply that a test case must be kept, and that it can be dropped, it will remain in the test set.

The last step in the incremental test selection process is to add new test cases. New test cases must be added for methods that are added, or the semantics or the interface of which is modified. However, since methods are tested in integration, it is also necessary to select from methods, the interface and the semantics of which is kept, a set of methods that must be tested in integration with the former methods. Notwithstanding the reason that to select meaningful test cases for the added and redefined methods, other methods must also be included in the test cases, the kept methods must also be included because of the influence that those added and redefined methods may have on their behavior: even if the domain space of attributes is not changed, it is possible that new methods reach states that could not be reached by a combination of the methods of the parent.

This set of methods can be calculated by several ways:

- A first criterion is encapsulation: depending on whether a descendant has visibility on the state of its parent, properties can be considered as alterable or not.
- A second criterion is state classification. Thus, it is possible to restrict the kept methods to test in integration to those that appear in the same set as the modified and added methods.
- A third criterion is use of attributes. If the same type of conformance hypothesis as above apply, it could be possible to eliminate the methods that have no effect on the attributes, the observer methods.

Those hypotheses can possibly be verified by a static analysis of the program.

7. Conclusion

The benefits of object-oriented software development methodologies can be substantial. However, using these methodologies does not guarantee correctness. We therefore consider essential the development of for verification and validation methodologies.

In this paper, we have presented an approach for specification-based testing of object-oriented software. It is based on refining a testing context into a practicable test set that preserves the pertinence of the exhaustive test set. We described a practical process for test case selection, based on the use of hypotheses. First, we presented several regularity hypotheses, which are *n:m* generalizations of the unit behavior and are expressed are used to select the shape of test formulae. We introduced constraints for regularity hypotheses of three kinds: constraints based on events, on the structure of objects and on the *HML* formulae themselves. We have also shown how the variables not fixed by those constraints can be instantiated using uniformity hypotheses, which are *1:n* generalizations of the unit behavior. We have also shown how the quality of the test set can be enhanced by performing subdomain decomposition.

To benefit from subtyping and subclassing, we have introduced a model that allows considering hierarchies at a finer level of granularity: inheritance of interface, of semantics, and of implementation. We have shown what test cases devised for a parent class can be reused for the descendant, and what new test cases must be added. This is achieved by a difference function that calculates what properties are removed, kept, modified and added from a parent to a descendant.

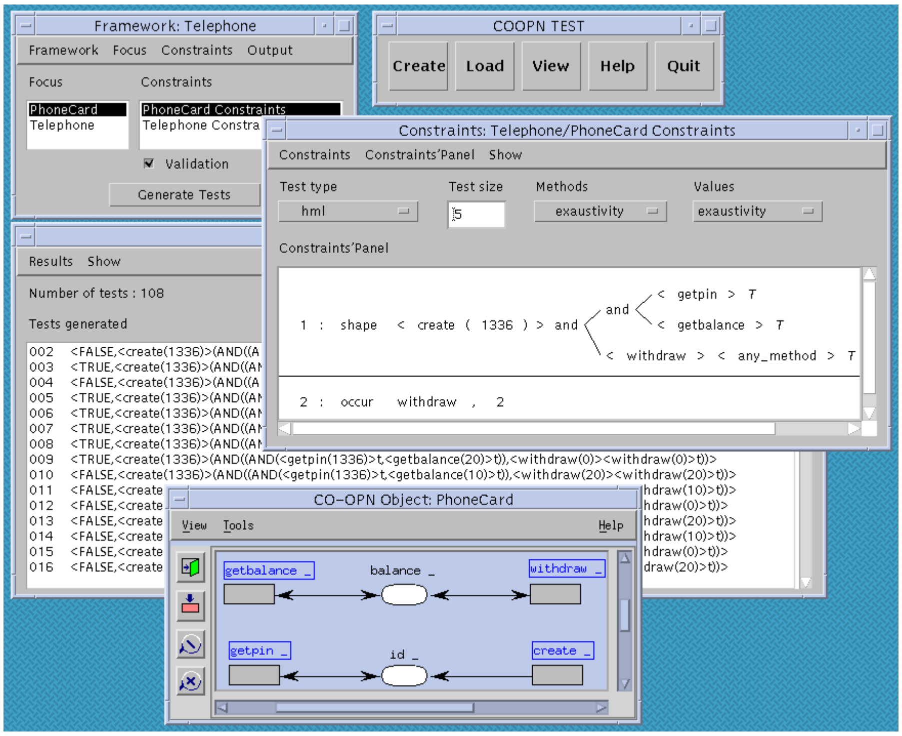

Fig. 6. Snapshot of *CO-OPNTest*

Our methodology is formal enough to semi-automate test selection. A Prolog tool, called *CO-OPNTest*, is being implemented, by translating 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 from the constraints presented in this paper. A front-end, written in Java, allows a user-friendly definition of the constraints. The figure 6 displays the snapshot of the tool. 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 depicts a graphical representation of the Petri nets of the objects in the system. This representation offers an intuitive comprehension of the specification to the tester and thus guides him during the test selection process.

We think that our approach to testing object-oriented software is promising. It has very strong theoretical foundations, it is general, and, notwithstanding the formal justification, it is sufficiently intuitive to be understood by the average programmer. It has both a theoretical and a practical value, and we can hope that it is a useful contribution to the validation and verification of object-oriented software.

8. References

[1] A. Arnould. *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, Jan. 1997.

[2] B. Balfour. On 'unit testing' and other uses of the term 'unit'. In *MCC '88 Military Computing Conference*, pages 127–130. Military Computing Institute, May 1988.

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

[4] S. Barbey, D. Buchs, and C. Péraire. 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, Oct. 1996. Springer verlag. Also available as Technical Report (EPFL-DI-LGL No 96/163).

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

[6] O. Biberstein, D. Buchs, and N. Guelfi. COOPN/2: A concurrent object-oriented formalism. In H. Bowman and J. Derrick, editors, *Proc. Second IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 57–72, Canterbury, UK, Mar. 1997. Chapman and Hall, London.

[7] D. Buchs and N. Guelfi. A concurrent object-oriented Petri nets approach for system specification. In M. Silva, editor, *12th International Conference on Application and Theory of Petri Nets*, pages 432–454, Aahrus, Denmark, June 1991.

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

[9] S. P. Fiedler. Object-oriented unit testing. *Hewlett Packard Journal*, 40(1):69–74, Apr. 1989.

[10] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, 1(2):156–173, June 1975.

[11] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick. Incremental testing of object-oriented class structures. In *Proceeding of the 14th international conference on Software Engineering*, pages 68–79,

Melbourne, Australia, May 11-15 1992. ACM Press.

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

[13] P. C. Jorgensen and C. Erickson. Object-oriented integration testing. *Communications of the ACM*, 37(9):30–38, Sept. 1994. Special issue on testing object-oriented software.

[14] S. Kirani. *Specification and Verification of Object-Oriented Programs*. PhD thesis, University of Minnesota, Dec. 1994.

[15] J. D. McGregor. Construction functional test cases using incrementally derived state machines. 1996.

[16] G. J. Myers. *The Art of Software Testing*. Business Data Processing: a Wiley Series. John Wiley & Sons, 1979.

[17] C. Péraire, S. Barbey, and D. Buchs. Test selection for object-oriented software based on formal specifications. Technical Report 97/252, EPFL-DI-LGL, Dec. 1997. Accepted in PROCOMET'98, Published in DeVa second year report-January 98.

[18] D. E. Perry and G. E. Kaiser. Adequate testing and object-oriented programming. *Journal of Object-Oriented Programming*, 2(5):13–19, Jan. 1990.

[19] P. Thévenod-Fosse and H. Waeselynck. Towards a statistical approach to testing object-oriented programs. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS-27)*, pages 99–108, Seattle, Washington, USA, June 1997.

[20] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, Dec. 1997.