A Theory of Specification-Based Testing for Object-Oriented Software

Stéphane Barbey, Didier Buchs and Cécile Péraire

Swiss Federal Institute of Technology
Software Engineering Laboratory
1015 Lausanne Ecublens
Switzerland

email: {Stephane.Barbey, Didier.Buchs, Cecile.Peraire} @ di.epfl.ch
phone: +41 (21) 693.52.43 - fax: +41 (21) 693.50.79

ABSTRACT The current strategies for testing object-oriented software all lack the formal basis which is necessary to perform this task efficiently. We propose the adaptation to object-oriented software of an existing theory of testing for stateless *ADTs*, to find errors in a class by checking that its implementation meets its specification. We present shortly in an informal way an object-oriented language, *CO-OPN/2*, in which language we will write the specification. We introduce a notion of test that takes into account the possible and impossible sequences of call of class methods. We examine the black-box test procedure, and give techniques to select a finite and pertinent test set from an exhaustive test set, including all the possible behaviors of the class under test, by applying test reduction hypothesis. We also study the construction of an oracle, the procedure that analyses the results of the tests, adapted to object-oriented software.

KEYWORDS object-orientation, testing, black-box testing strategy, formal methods, validation and verification, test and evaluation.

1 Introduction

Object-oriented methods are now used for all kinds of software developments. Although some people have assumed that object-orientedness leads by itself to quality, experience has proved that object-oriented software cannot escape to a validation and verification process. One way to verify that a program meets its specification is testing.

In this paper, we will focus on specification-based (also known as black-box) testing. This kind of testing is usually a three steps process. It starts with a given requirement, to find faults in a program by comparing it against a specification. It is decomposed in a *test selection step*, in which the test cases that will validate or not the requirement are generated, a *test execution step*, in which the test cases are executed and results of the execution collected, and a *test satisfaction step*, in which the results obtained during the test execution phase are compared to the expected results. This last step is commonly performed through the help of an oracle [12]. The results of this comparison give the result of the requirement, 'yes' if it is validated, 'no' it is not, and 'undecided' if the oracle is unable to analyse the result, or if the test execution did not return a result. If the answer is 'no', then the test set has found errors in the program —the goal of testing—, and the programmer can correct the implementation to remove them. Of course, the main problem is to build a test set that has enough significance to find a maximal number of errors, so that a 'yes' answer gives confidence to the programmer that the program meets its specification.

It has been argued, for example by [11], that testing object-oriented software is very similar to testing traditional software, or that it is even simplified, and that it does not require specific strategies. (By traditional software development, we understand functional decomposition development.) However, testing must take into account the specifics of the object-oriented developments methods, and of the structure of object-oriented software. Traditional strategies need adaptation to be fit for object-oriented systems. Several strategies for testing object-oriented software have already been proposed. However, those strategies lack the theoretical basis necessary to ensure the quality of the development, and especially the meaning of the 'yes' answer. In this paper, we propose a generalization and an adaptation to object-oriented software of the Bernot, Gaudel, and Marre theory of testing (called *BGM* theory), described in [3]. This theory considers testing of abstract data types. We enhance it to meet the specifics of object-oriented systems. This theory is formally defined and we will keep this rigor in our adaptation. An other advantage of this theory is that it is fit for automatization. A tool, called *LOFT* [9], has already been implemented for algebraic specifications and an adaptation of *LOFT* to the presented theory is planned.

The structure of the paper is the following. In section 2, we present the syntax and the semantics of a concurrent object-oriented specification language, called *CO-OPN/2* (Concurrent Object-Oriented Petri Nets), using a small but significant example: a phonecard system. In section 3, we propose a theory of specification-based testing for object-oriented software, introducing for each step of the test procedure both the general theory of testing and the adaptation to the particular case of object-oriented systems.

2 Specifying Object Systems

Although the theory presented in this paper does not require the specification of the software under test to be written using a formal specification language, the use of such a language is beneficial to ensure the consistency and the completeness of the specification. In this section, we will briefly present a concurrent object-oriented specification language, called *CO-OPN/2*. Interested readers will find more complete information in [4] and [5].

CO-OPN/2 (Concurrent Object-Oriented Petri Nets) is a formalism developed for the specification and design of large object-oriented concurrent systems. Such a system consists of a possibly large amount of entities, that communicate with each other by sending messages, or, in our (concurrency-friendly) terminology, by triggering parameterized events. The events to which an object can react are also called its methods. The behavior of the system is expressed with a Petri net, and the semantics of this system is given through a transition system.

Some objects in the system are active objects. The synchronization on the active objects is specified by means of synchronization expressions that appear in the axioms of the classes. To communicate with an instance of a class (an object), i.e. to send it a message, a client object must synchronize itself with that object by means of triggering one of its external events.

2.1 Architecture of a *CO-OPN/2* Specification

A *CO-OPN/2* specification consists of a collection of two different kinds of modelling entities: algebraic abstract data types (*ADTs*) and classes. Two different kinds of sorts, algebraic sorts and class' type sorts (later simply called types), are defined in those entities.

- Algebraic sorts are defined together with related operations, called functions, in *ADT* modules. A variable of an algebraic sort has no identity. Algebraic sorts are used to specify values such as the primitive sorts (integer, boolean, enumeration types, etc.) or purely functional sorts (stacks, etc.).
- Each class' type sort and its bound operations —its methods— are defined in one class module. Such a module corresponds to the notion of encapsulated entity that holds an internal state and provide the outside with various methods. A class module is a template from which objects can be instantiated, and that contains a type definition, whose variables will have an identity, defined by an object identifier algebra. Class types are used to define the classes in an object-oriented system.

To illustrate our theory, we introduce a small but significant example: a simplified phonecard system. This example models the behavior of a telephone machine that can be used with a phonecard. Each phonecard includes an identification code (the *ADT* Pin), that is also known to the user, and requested by the telephone machine when the user tries to make a phone call and to pay with the card. The remaining balance is also written on the card. We model this system using several *ADTs* (Pin, Money, Bool and Nat) and two classes (PhoneCard and Telephone).

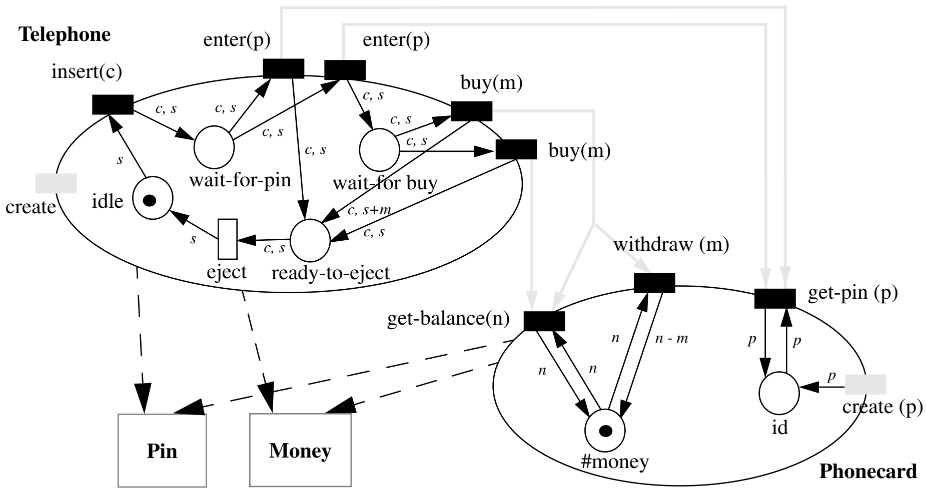

Fig. 1. The classes and their internal description and synchronizations

In figure 1, we give a partial representation of the classes, their synchronizations and their clientship relations with abstract data types. Inside each oval, representing an encapsulated class, one Petri net describes the behavior of the considered class. This Petri net represents the state of the objects through the places (circles) and the object methods through the external transitions (black rectangles). Internal transitions (the usual transitions of Petri nets) describe the internal invisible behavior (white rectangles) hidden in the semantics. Creation transitions appear as grey rectangles. Synchronizations between transitions are drawn using grey arrows.

2.2 The Syntax of a *CO-OPN/2* Specification

Every modelling entity (*ADT* or class) has a *header* (which holds the module name and the information related to genericity or inheritance), an *interface* (corresponding to the visible part of a module; it includes the information on the components of the module that are accessible to its clients) and a *body* (that describes the local aspects of the module, such as the behavior of its operations, by means of axioms).

In our example, the balance (*ADT* Money) as well as the personal identification number (*ADT* Pin) are static objects and then are correctly modelled as abstract data types (see figure 2). The *ADTs* Bool and Nat are standard data types axiomatized as usual. Since the *ADT* Money is defined to model balance, it seems natural to describe this type as a renamed copy of the *ADT* Nat. It inherits all the operations of Nat. The *ADT* Pin is defined with two generators to create new pins and one operation to compare two pins in order to validate the access to a phonecard.

ADT Pin;
    Interface
        Use Bool;
        Sorts pin;
        Generators
            first-pin : -> pin;
            new-pin _ : pin -> pin;
        Operations
            _=_ : pin pin -> bool;
    Body
        Axioms
            (new-pin n = new-pin m) = (n = m);
            (new-pin n = first-pin) = false;
            (first-pin = new-pin m) = false;
            (first-pin = first-pin) = true;
        Where n, m : pin;
End Pin;
	ADT Money **as** Nat;
	    Renaming nat -> money;
	End Money;

Fig. 2. The *ADTs* Pin and Money

Cooperation between objects is performed by means of a synchronization mechanism, i.e. each event may request synchronization with the method 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

in which

- Condition is a condition on the 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 at the different places within the net (in arcs and out arcs in the net).

For instance, the figure 3 shows the textual description of the class Phonecard. (The graphical representation in figure 1 is equivalent to that textual representation). The state of a phonecard is described by the place #money, which stores the money still available on the card, and id, which stores the associated 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 (get-pin), to access the balance (get-balance), and to reduce it (withdraw). Since it is a throw-away card, it is not possible to reload it. In the field **Axioms**, the behavior of the methods is given by Petri net causal relations. Since the places are independently accessed through the methods, simultaneous access can be performed on the pin and on the balance. However, it is not possible to simultaneously read the balance and withdraw it because get-balance and withdraw used the same critical resource #money.

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

Fig. 3. Textual specification of the class Phonecard

The class Telephone (figure 4) specifies the behavior of the automaton which accepts a card, waits for and checks a pin number, and, as long as the pin is correct, reduces the balance of the card of a given amount corresponding to the price of a phone call. These operations are events corresponding to the methods insert, enter and buy, which can be activated sequentially. The places of the class are used to describe the state of the telephone. In this state figures the money already collected by the telephone, and, when a card is in the machine, a reference to this card. These data are also used to sequentialize the events which are possibly activated. The internal transition eject is automatically activated if an error occurs, or when the process is finished. Its effects are not directly observable, but correspond to the rejection of the card. In the class header, the field Object allows the definition of statically created objects, in our example an object called cabin, which can be referenced as an object of the class.

```
Class Telephone;
    Interface
        Use Pin, Money, PhoneCard;
        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;
```

Fig. 4. Textual specification of the class Telephone

2.3 The Semantics of a *CO-OPN/2* Specification

In a *CO-OPN/2* specification, all the possible behaviors can be modelled by a labelled transition system. A labelled transition system $G = \langle Q, E, \rightarrow, I \rangle \in \Gamma$ (where Γ is the class of all transition systems) is such that:

- Q: set of states (built over the local state of each object)
- E: set of events (built from objects creations, calls of methods and algebraic terms)
- $\rightarrow \subseteq Q \times E \times Q$: transition relationship (noted: $q \xrightarrow{e} q'$)
- $I \in Q$: a non-empty initial state

These models are built for *CO-OPN/2* using inference rules which are not given here due to the lack of space, interested readers should find the detailed descriptions in [4] and [5]. We will see (cf. full agreement theorem) that a labelled transition system must be image-finite, this will be a hypothesis of the testing procedure:

- A state $q \in Q$ is image-finite, if $\{q' \in Q \mid q \xrightarrow{e} q'\}$ is finite for each $e \in E$.
- G is image-finite, if all the reachable states of G are image-finite.

A program $P \in$ *PROG* (*PROG*: the class of all programs) is said to have the same semantics as a specification $SP \in$ *SPEC* (*SPEC*: the class of all specifications), if P satisfies to a satisfaction relationship $\models$. This relationship is defined with regard to the bisimulation equivalence ($\underline{\leftrightarrow}$) which identifies systems with a similar arborescent structure. By bisimulation we understand *strong* bisimulation which is the adequate equivalence relationship for CO-OPN/2, because the semantics of CO-OPN/2 hide the internal transitions. Readers interested in the bisimulation equivalence relationship should refer to [10]. We often use in the next definitions the function *EVENT*(*X*) which returns all the events of a system *X* expressed in a considered language.

Definition: Satisfaction relationship $\models$

Let $P \in$ PROG a program, $SP \in$ SPEC its *CO-OPN/2* specification, $G(P) = \langle Q_1, EVENT(P), \rightarrow_1, I_1\rangle$ and $G(SP) = \langle Q_2, EVENT(SP), \rightarrow_2, I_2\rangle$ two labelled transition systems modeling *P* and *SP*, where the sets of events *EVENT*(*P*) and *EVENT*(*SP*) are the same. Assuming *P* and *SP* have a similar signature, the satisfaction relationship $\models \subseteq PROG \times SPEC$ is such that:

$$(P \models SP) \Leftrightarrow (G(P) \underline{\leftrightarrow} G(SP)).$$

The test procedure is based on the knowledge of 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 the desired implementation satisfy. In practice it is not necessary to have the same language to express the specification properties and the tests. The most interesting solution is to have a specification language well adapted to the expression of properties from an user point of view, and an other 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.

If the specification language is *CO-OPN/2*, the tests could be expressed with the *HML* Logic (or with any other logic compatible with $\models$), introduced by Hennessy-Milner in [7], because, in the context of image-finite labelled transition systems, it exists a full agreement between the bisimulation equivalence ($\underline{\leftrightarrow}$) and the *HML* equivalence ($\sim_{HML}$). Hence the following *HML* definitions and full agreement theorem:

Definition: Syntax of *HML*

The HML_{SP} language is defined for a specification $SP \in$ SPEC as follows:

- $T \in HML_{SP}$
- $f \in HML_{SP} \Rightarrow (\neg f) \in HML_{SP}$
- $f, g \in HML_{SP}\ P \Rightarrow (f \wedge g) \in HML_{SP}$
- $f \in HML_{SP} \Rightarrow (\langle e\rangle f) \in HML_{SP}$ where $e \in EVENT(SP)$

Definition: Semantics of *HML*

Given $SP \in$ SPEC a specification, $G = \langle Q, EVENT(SP), \rightarrow, I\rangle$ a labelled transition system with same events as the specification, and $q \in Q$. The satisfaction relationship $\models_{HML_{SP}} \subseteq \Gamma \times Q \times HML_{SP}$ is defined as follows:

- $G, q \models_{HML_{SP}} T$
- $G, q \models_{HML_{SP}} (\neg f) \Leftrightarrow G, q \not\models_{HML_{SP}} f$
- $G, q \models_{HML_{SP}} (f \wedge g) \Leftrightarrow G, q \models_{HML_{SP}} f$ and $G, q \models_{HML_{SP}} g$
- $G, q \models_{HML_{SP}} (\langle e\rangle f) \Leftrightarrow$ it exists $e \in EVENT(SP)$ such that $q \xrightarrow{e} q'$ with $G, q' \models_{HML_{SP}} f$

Let $f \in HML_{SP}$ a formula, we write $G \models_{HML_{SP}} f$ when $G, I \models_{HML_{SP}} f$.

Definition: *HML* Equivalence

Given $SP \in$ *SPEC* a specification, $G_1 = \langle Q_1, EVENT(SP), \rightarrow_1, I_1\rangle$ and $G_2 = \langle Q_2, EVENT(SP), \rightarrow_2, I_2\rangle$ two labelled transition systems, the HML_{SP} equivalence relationship ($\sim_{HML_{SP}}$) is such that:

$$(\forall f \in HML_{SP}, G_1 \models_{HML_{SP}} f \Leftrightarrow G_2 \models_{HML_{SP}} f) \Leftrightarrow (G_1 \sim_{HML_{SP}} G_2).$$

Theorem: Full Agreement

Given $SP \in$ *SPEC* a specification, $G_1 = \langle Q_1, EVENT(SP), \rightarrow_1, I_1\rangle$ and $G_2 = \langle Q_2, EVENT(SP), \rightarrow_2, I_2\rangle$ two image-finite labelled transition systems, we have:

$$(G_1 \underline{\Leftrightarrow} G_2) \Leftrightarrow (G_1 \sim_{HML_{SP}} G_2).$$

An advantage of this approach is to have an observational description of the valid implementation through the tests. One test is a formula which is valid or not in the specification and that must be experimented in the program i.e. correct implementation behaves similarly on the tests. Other approaches can be taken, depending on the hypothesis we have select about the valid implementation, for instance traces are used as test patterns for ASTOOT [6].

3 The Theory of Testing

The "black box" test method is an approach to find errors in a program by validating its functionalities, without analysing the details of its code, but by using the specification of the system. The goal is to answer the question: *"Does a program satisfy the requirements of its specification?",* or, in accordance to the goal of testing, to find if a program does not satisfy its specification. We will study in the following the relationships between the specification of a system and its realizations. Through the help of the oracle (formally denoted by $\models_O$) we will decide if the tests that are produced by the examination of the specification are successful or not. The final result will be that the program is acceptable or not for the given specification.

3.1 The Test Procedure

The strategy used to answer to this former question is to select, from the specification, the services required from the system. For each service, the specification allows the selection of a number of scenarios for the program under test. The set of all these scenarios makes up what we call the test set. The test procedure is as follows:

Given:

- *SPEC*: class of all specifications written in the specification language considered,
- *PROG*: class of all programs expressed in the language used for the implementation,
- *TEST*: class of all test sets that can be written,
- $\models$: satisfaction relationship on *PROG* × *SPEC*, expressing the validity of the program with respect to the specification,
- $\models_O$: satisfaction relationship on *PROG* × *TEST*, deciding if the tests are successful or not for the program under test.

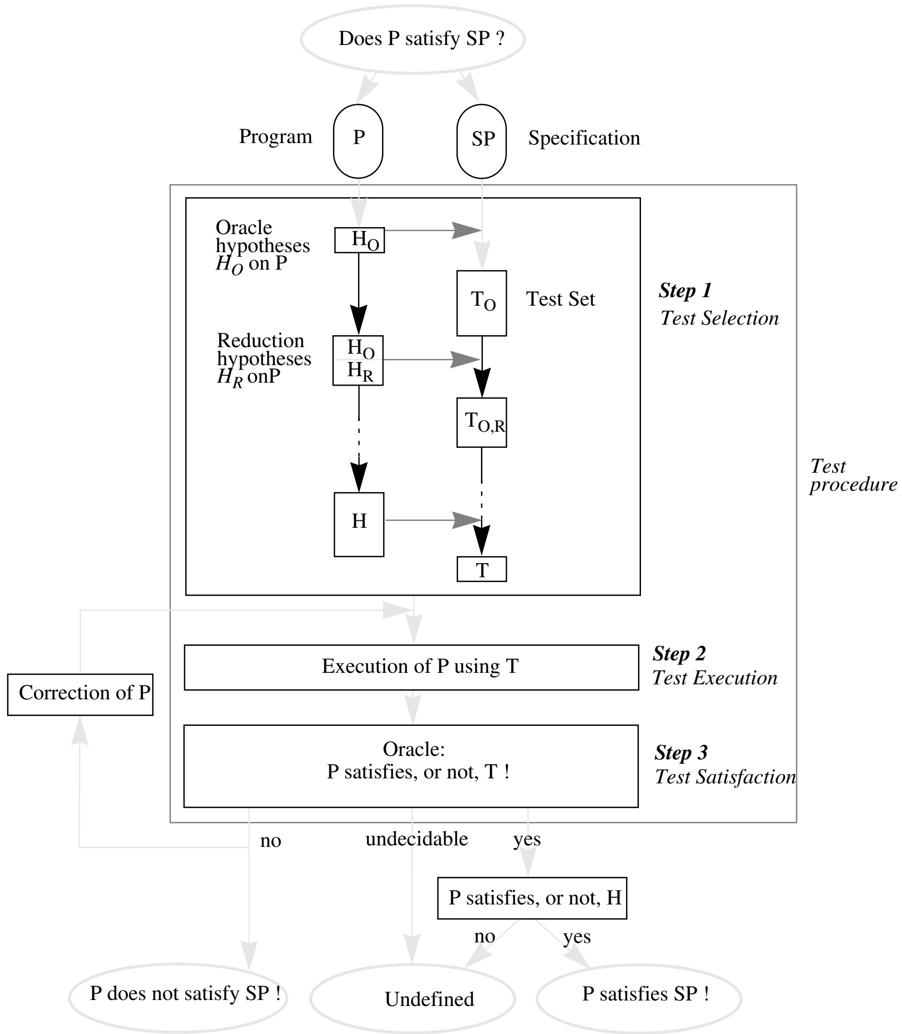

Fig. 5. Test procedure

Step 1 Selection of a test set $T_{SP,\,H}$ from a specification SP of the system and from a set of hypotheses H on the program under test P.

$$T_{SP,\,H} \in \textit{T}\small{EST}, SP \in \textit{S}\small{PEC}, P \in \textit{P}\small{ROG}.$$

Step 2 Execution of the program under test P using the test set $T_{SP,\,H}$.

Step 3 Analysis of the results obtained during the execution of the program P.

If the test set is successful ($P \models_O T_{SP,\,H}$), then the test procedure is finished. In addition, if P verifies the hypotheses H, then the program satisfies the requirements of the specification ($P \models SP$). If the test set is not successful, then the program contains faults or omissions, and it is possible to return to the second step of the test procedure after having corrected P. Note that the test procedure itself is successful if the test set helped finding as many errors as possible, i.e. if the test set is unsuccessful. The efficiency of the test set, i.e. its aptitude to detect errors, must be aligned to some quality criteria. For instance, the realization of an aerospace system requires a higher level of quality than the realization of a computer aided design system.

The test procedure is shown on figure 5. These steps are discussed in more detail by the following text. To implement the procedure of the “black box” test method it is necessary to answer the following questions:

- *“How should we select a test set?”*,
- *“How should we determine the success or the failure of a selected test set?”*.

The test set selection method is based on the existence of decision criteria for the success or failure of a test. To answer the previous questions, it is necessary to start with the second question, assuming that a test set has been selected during the first step of the test procedure. Beforehand, we will particularize the theory of testing to the particular case of object oriented systems.

3.1.1 A Theory for Testing Object-Oriented Software

In the object-oriented context, an elementary test for a program under test $P \in$ PROG and a specification $SP \in$ SPEC can be defined as a couple <*Formula*, *Result*> where:

- *Formula*: (ground) temporal logic formula, such that *Formula* belongs to HML_{SP}.
- *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 <*Formula*, *Result*> is successful if *Result* is *true* and *Formula* is valid in the (image-finite) labelled transition system modeling the expected behavior of P, or if *Result* is *false* and *Formula* is not valid in the (image-finite) labelled transition system modeling the expected behavior of P. In all other cases, a test <*Formula*, *Result*> is a fails. It is important to note that the test definition will allow the test procedure to verify that a non-acceptable scenario cannot be produced by the program (ex: make a call even though the identification code of the phonecard is wrong).

Formula can be expressed with the *HML* logic. For example, given G the labelled transition system modeling the telephone in figure 1 and a phonecard referenced by the name card and containing the pincode 1234 obtained with the following creation sequence <card.create 1234> (1234 is the decimal representation of new-pin applied 1234 times to first-pin) and the initial balance 20, we have:

$G \models_{HML_{Telephone}}$ <cabin.create> <cabin.insert card> <cabin.enter 1234> <cabin.buy 12> T

because making a call is possible when the identification code is right.

$G \not\models_{HML_{Telephone}}$ <cabin.create> <cabin.insert card> <cabin.enter 4321> <cabin.buy 12> T

because making a call is impossible when the identification code is wrong.

3.2 The Oracle

Once a test set has been selected, it is used during the execution of the program under test. Then the results collected from this execution must be 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.

Definition: Oracle

The oracle $O = \langle \models_O, Dom_O \rangle$ is a partial decision predicate of a formula in a program P. For each test $\phi \in$ *TEST* belonging to the oracle domain Dom_O, the satisfaction relationship $\models_O$ on *PROG* $\times$ *TEST* allows the oracle to decide:

- If ϕ is successful in P ($P \models_O \phi$).
- If the answer is undefined (for instance, a deadlock situation which prevents the normal termination of P).

The oracle is constituted of a collection of equivalence relationships which compare similar elements of the scenario derived from the specification to the program under test; these elements are said to be observable. The problem is that the oracle is not always able to compare all the necessary elements to determinate the success or failure of a test; these elements are said to be non-observable. This problem is solved using the oracle hypotheses H_O which are part of the possible hypotheses and collect all power limiting constraints imposed by the realization of the oracle:

Definition: Oracle Hypotheses

The oracle hypotheses H_O are defined as follows:

- When a test $\phi \in$ *TEST* is observable ($\phi \in Dom_O$) for a program P, the oracle knows how to decide the success or the failure of ϕ:

$$(P \text{ satisfies } H_O) \Rightarrow ((P \models_O \phi) \vee (\neg (P \models_O \phi))).$$

- When a test ϕ is non-observable for a program P ($\phi \in Dom_O$), the oracle has a set C of criterion c_i allowing to observe ϕ:

$$(P \text{ satisfies } H_O \wedge P \models_O (\wedge_{c_i \in C}\, c_i (\phi))) \Rightarrow (P \models_O \phi).$$

It seems rational to put the oracle hypotheses H_O, which assure the observability of the system, at the beginning of the selection test set procedure (see figure 5).

The oracle evaluation is performed using the equality for functional approaches and traces for concurrency or state based approaches. For object oriented languages, and for *CO-OPN/2*, the oracle must handle formulae such as process logic or traces of method calls, to take into account the state of the objects.

3.2.1 Oracle for Object Specifications

In the case of object systems, the observation criteria of a test formula expressed using *HML* logic, which is operational by nature, can be (with a simple oracle) the paths of the formula. A path is a subformula without *And* operators. These paths are submitted to the program under test that accepts or rejects them. Acceptation or rejection are compared to the expected behavior expressed in the test by the boolean value *Result* attached to the logical formula. If they correspond, the test is successful, otherwise it reveals an error. A more sophisticated oracle should introduce state memorization to compute the *And HML* operator. This is an intrusive way of implementing the oracle. See [8] for an explanation of the *CO-OPN/2* operational semantics.

Definition: Satisfaction relationship $\models_O$

Let $P \in$ *PROG* a program, $SP \in$ *SPEC* a specification, $G(P) = \langle Q, EVENT(P), \rightarrow, I \rangle$ a labelled transition system modeling the expected behavior of P such that $EVENT(SP) = EVENT(P)$, $T_{SP,\,H} \in$ *TEST* a test set, $Formula \in HML_{SP}$ and $Result \in \{true, false\}$. The satisfaction relationship $\models_O \subseteq$ *PROG* $\times$ *TEST* is such that:

$$(P \models_O T_{SP,\,H}) \Leftrightarrow$$
$$(\forall \langle Formula, Result\rangle \in T_{SP,\,H}\ ((G(P) \models_{HML_{SP}} Formula \text{ and } Result = true) \text{ or}$$
$$(G(P) \not\models_{HML_{SP}} Formula \text{ and } Result = false))).$$

An oracle hypothesis of H is that the labelled transition system modelling the expected behavior of P is image-finite. This property is necessary to state the theorem of correspondence between the test satisfaction and the specification validity. Moreover, another necessary oracle hypothesis is the assumption of a bounded and fair non-deterministic call of methods. It involves making a limited number of applications of the same test.

3.3 The Test Set Selection Problem

Assuming that we have an oracle O that ensures the observability of the system with the oracle hypotheses H_O, the first task of the test procedure consists of selecting, from the specification, a test set that allows the exhaustive validation of each service required from the system. This is theoretically achieved by selecting an exhaustive test set which contains all the test that are required by the specification. However the exhaustive test set selected is generally infinite, and it is necessary to apply a number of reduction hypotheses H_R to the behavior of the program to obtain a finite test set of "reasonable" size. Therefore, we proceed by successive reductions of the number of tests (see figure 5). Thus, when the test set is successful, the program is correct on condition that it satisfies the oracle and the reduction hypotheses. The test set quality is therefore function of the number of oracle and reduction hypotheses satisfied by the program under test. The idea of the test set selection procedure is to find $T_{SP,\,H}$ such that:

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

The equivalence relationship is satisfied when the test set $T_{SP,\,H}$ is pertinent, i.e. valid (any incorrect program is discarded) and unbiased (it rejects no correct program):

Definition: Pertinent Test Set

Given H hypotheses, $P \in$ *PROG* and a specification $SP \in$ *SPEC*. The test set $T_{SP,H} \subseteq$ *TEST* is pertinent iff:

- $T_{SP,\,H}$ is valid: $(P \text{ satisfies } H) \Rightarrow (P \models_O T_{SP,\,H} \Rightarrow P \models SP)$.
- $T_{SP,\,H}$ is unbiased: $(P \text{ satisfies } H) \Rightarrow (P \models SP \Rightarrow P \models_O T_{SP,\,H})$.

A pertinent test set $T_{SP,\,H}$ can be used to test a program P only if $T_{SP,\,H}$ has a "reasonable" finite size. For that purpose we build a test context, called practicable because it can be effectively applied to the oracle, using reduction hypotheses. Beforehand, we will particularize the theory of testing to the particular case of object oriented systems.

3.3.1 The Test Selection for Object Oriented Specifications

In this section, we give the relationship between the two satisfaction relationships ($\models$ and $\models_O$) by using a corollary of the full agreement theorem (see section 2.3). For that purpose, we define the exhaustive test set.

Definition: Exhaustive Test Set

Given $SP \in \textit{SPEC}$ a specification, $G(SP) = \langle Q, EVENT(SP), \rightarrow, I \rangle$ an image-finite labelled transition system representing the semantics of the object system specification SP and H_O the oracle hypotheses. An exhaustive test set $\textit{EXHAUST}_{SP,\ Ho} \subseteq \textit{TEST}$ is such that:

$$\begin{aligned} \textit{EXHAUST}_{SP,\ Ho} = \{ & \langle Formula, Result \rangle \in HML_{SP} \times \{true, false\} \mid \\ & (G(SP) \models_{HML_{SP}} Formula \text{ and } Result = true) \text{ or} \\ & (G(SP) \not\models_{HML_{SP}} Formula \text{ and } Result = false) \}. \end{aligned}$$

Corollary of the full Agreement Theorem:

Let $P \in \textit{PROG}$ an object-oriented system under test, $SP \in \textit{SPEC}$ its specification, and $\textit{EXHAUST}_{SP,\ Ho}$ an exhaustive test set obtained from SP and from a set of hypotheses H_O on P. We have:

$$(P \text{ satisfies } H_O) \Rightarrow (P \models SP \Leftrightarrow P \models_O \textit{EXHAUST}_{SP,\ Ho}).$$

The figure 6 summarizes the correspondences between the different theoretical relationships, on which the test process is built, introduced in this paper.

$$\begin{array}{ccccc} & & \text{Full Agreement Theorem} & & \\ & \underline{\leftrightarrow} & \Leftrightarrow & \sim_{HML} & \\ \text{Definition} & \Updownarrow & & \Updownarrow & \text{Definition} \\ & \models & \Leftrightarrow & \models_O & \\ & & \text{Full Agreement Corollary} & & \end{array}$$

Fig. 6. Full agreement theorem and its corollary

For the example of the telephone cabin and the phonecard card containing the pincode 1234 and the initial balance 20, the exhaustive test set contains tests such as:

<<cabin.create> <cabin.insert card> <cabin.enter 1234> <cabin.buy 12> *T*, *true*>

<<cabin.create> <cabin.insert card> <cabin.enter 4321> <cabin.buy 12> *T*, *false*>(1)

<<cabin.create> <cabin.insert card> <cabin.enter 1234> <cabin.buy 12>
<cabin.insert card> <cabin.enter 1234> <cabin.buy 6> *T*, *true*>

The two first tests correspond to an excerpt of the possible combinations of events applied in one single cycle of telephone usage, whereas the third corresponds to successive uses of the cabin. We can also give more sophisticated tests, including the $\wedge$ (and) and $\neg$ (negative) operators, such as:

<<cabin.create> <cabin.insert card> <cabin.enter 4321> ¬(<cabin.buy 12> *T*), *true*>

<<cabin.create> <cabin.insert card> (<cabin.enter 1234> <cabin.buy 12> *T*) $\wedge$
(<cabin.enter 4321> <cabin.buy 12> *T*), *false*>

The last test is redundant with respect the test (1). In the following section, we will explain how to avoid such redundancy, while keeping the pertinence of the test set, by reducing the exhaustive test set using adequate strategies.

3.4 Construction of a Practicable Test Context and Reduction Hypotheses

We express in this section how to iteratively reduce a test set, in order to reach a test set that reflects the expected behaviors that are crucial for the validation of an implementation with respect to its specification. The definition of a practicable test context is the following:

Definition: Practicable Test Context

Given a specification $SP \in$ *SPEC*. A test context $(H, T_{SP,\,H})_O$ is defined by a set of hypotheses H on a program under test P, a test set $T_{SP,\,H} \subseteq$ *TEST* and an oracle O.

$(H, T_{SP,\,H})_O$ is practicable iff:

- $T_{SP,\,H}$ is pertinent and has a “reasonable” finite size.
- $O = < \models_O, Dom_O>$ is defined for each element of $T_{SP,\,H}$ ($T_{SP,\,H} \subseteq Dom_O$).

To simplify, $(H, T_{SP,\,H})_O$ is noted $(H, T)_O$ in the rest of the paper.

The selection of a pertinent test set $T_{SP,\,H}$ of “reasonable” size is made by successive refinements of an initial test context $(H_0, T^0)_O$ which has a pertinent test set $T^0_{SP,\,H_0}$, until the obtention of a practicable test context $(H, T)_O$:

$$(H_0, T^0)_O \leq \ldots \leq (H_i, T^i)_O \leq (H_j, T^j)_O \leq \ldots \leq (H, T)_O.$$

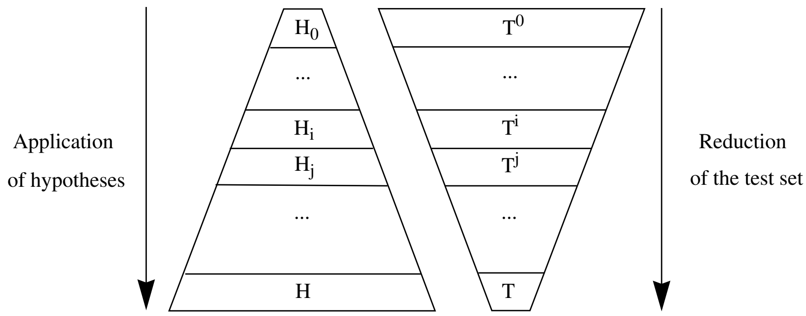

Fig. 7. Iterative refinement of the test context

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: $H_j \Rightarrow H_i$.
- The test set $T^j_{SP,\,H_j}$ is included in the test set $T^i_{SP,\,H_i}$: $T^j_{SP,\,H_j} \subseteq T^i_{SP,\,H_i}$.
- If P satisfies H_j then $(H_j, T^j)_O$ does not detect more errors than $(H_i, T^i)_O$:
  $$(P \text{ satisfies } H_j) \Rightarrow (P \models_O T^i_{SP,\,H_i} \Rightarrow P \models_O T^j_{SP,\,H_j}).$$
- If P satisfies H_j then $(H_j, T^j)_O$ detects as many errors than $(H_i, T^i)_O$:
  $$(P \text{ satisfies } H_j) \Rightarrow (P \models_O T^j_{SP,\,H_j} \Rightarrow P \models_O T^i_{SP,\,H_i}).$$

Therefore, we have the following theorem:

Theorem:

Given specification $SP \in$ *SPEC*. Given two test contexts $(H_i, T^i)_O$ and $(H_j, T^j)_O$ such that $(H_i, T^i)_O \leq (H_j, T^j)_O$. If $T^i{}_{SP,\, H_i}$ is pertinent then $T^j{}_{SP,\, H_j}$ is pertinent.

A test is a formula with or without universally quantified variables. The reduction hypotheses act on tests with variables, replacing theses variables to reduce the formula complexity. Thus we define $Var(\phi)$: the set of variables of a formula ϕ.

3.4.1 The Uniformity Hypotheses

The uniformity hypotheses, which are very strong, stipulate that if a test of a formula ϕ, with $v \in Var(\phi)$, is successful for a given value of v, then it is successful for all possible values of v. For instance, in the case of a stack, if a test is successful for a stack which has a top element of 2, then it is successful for all possible values of the top element. Since the uniformity hypothesis is very strong, it is usually not applied alone to the sort under test, but to the sorts imported into the specification, which we suppose being already tested. Uniformity hypotheses can be combined with domain decomposition.

3.4.2 The Regularity Hypotheses

The regularity hypotheses stipulate that if a test of a formula ϕ, containing a term t, is successful for all terms t which have a complexity less or equal than a bound k, then it is successful for all possible complexities of t. For instance, in the case of an unbounded stack, if a test is successful for all stacks of a size smaller or equal to 20, then it is successful for all possible stack sizes.

In our case, if an uniformity hypothesis or a regularity hypothesis is applied on a pertinent test set, then the new test set obtained is pertinent too. Note that those two hypotheses correspond (possibly in a more formal way) to usual test practices. In further works, we will study the *HML* formula domain decomposition, and we will explore other interesting reduction hypotheses, such as incremental hypotheses, to build the test set of a descendant class upon the test set already selected for its parents.

3.4.3 Application of the Test Reduction Hypotheses for Object-Oriented Systems

The test reduction hypotheses can be applied to all three dimensions of a test: object management, *HML* formulae selection, algebraic term parameters selection.

- Object management

For the object management mechanism (class operations), both the regularity and the uniformity hypothesis can be used depending on the confidence on the underlying object management system which creates, destroys and modifies objects. If the system is supposed to work properly, stronger hypothesis may be chosen such as the uniformity hypothesis on the class type under test. Otherwise, if the system can have unexpected behavior depending on the number of objects, hypothesis such as the regularity must be applied.

For instance we can apply uniformity on the class Telephone, assuming that we have a phonecard referenced by card containing the pincode (new-pin first-pin) (obtained with the following creation sequence <card.create new-pin first-pin>) and the initial balance 20. This will produce the test set:

T_1 = {<<cabin.create> *f*, *result*> |
f ∈ *HML*$_{TELEPHONE}$, *result* ∈ {*true*, *false*} such that (s.t.) the test is valid}

Other test sets shall be applied using the regularity hypothesis if the number of telephone machine could produce wrong behavior.

- *HML* formulae

For *HML* formulae, the regularity hypothesis can be applied to reduce the size of the formulae to be tested, and the uniformity hypothesis can be applied on formulae of a given size in order to reduce the number of tests. Moreover, we can apply a reduction technique based on the semantics of tests, since some tests imply other tests. For instance redundant tests such as <*f*, *true*> and <¬(*f*), *false*> are equivalent and can be reduced to one test.

For instance, T_1 can be reduced by applying an uniformity hypothesis on the variable *f* producing the test set T_2 (resp. T_3):

T_2 = {<<cabin.create> <cabin.insert card> <cabin.enter *pin_1*> <cabin.buy *amount_1*>
<cabin.insert card> <cabin.enter *pin_2*> <cabin.buy *amount_2*> *T*, *result*> |
pin_1, *pin_2* ∈ pin, *amount_1*, *amount_2* ∈ money, *result* ∈ {*true*, *false*} s.t. the test is valid}

T_3 = {<<cabin.create> <cabin.insert card> (<cabin.enter *pin_1*> <cabin.buy *amount_1*> *T*) ∧
(<cabin.enter *pin_2*> <cabin.buy *amount_2*> *T*), *result*> |
pin_1, *pin_2* ∈ pin, *amount_1*, *amount_2* ∈ money, *result* ∈ {*true*, *false*} s.t. the test is valid}

T_3 (of complexity 6) covers all tests (of complexity 4) of the following pattern:

T_4 = {<<cabin.create> <cabin.insert card> <cabin.enter *p*> <cabin.buy *amount*> *T*, *result*> |
p ∈ pin, *amount* ∈ money, *result* ∈ {*true*, *false*} s.t. the test is valid}

which can be suppressed from the test set.

From T_1, a regularity hypothesis of complexity 2 can also be applied on the variable *f* producing the test pattern T_5.

T_5 = {<<cabin.create> <cabin.insert card> *T*, *result*>,
<<cabin.create> <cabin.enter *pin*> *T*, *result*>,
<<cabin.create> <cabin.buy *amount*> *T*, *result*>,
<<cabin.create> <cabin.insert card> <cabin.insert card> *T*, *result*>,
<<cabin.create> <cabin.insert card> <cabin.enter *pin*> *T*, *result*>,
<<cabin.create> <cabin.insert card> <cabin.buy *amount*> *T*, *result*>, ... |
pin ∈ pin, *amount* ∈ money, *result* ∈ {*true*, *false*} s.t. the test is valid}

- Algebraic term

Starting with a test set generated by the previous selection procedure we apply on the algebraic variables the regularity or the uniformity hypothesis and we fix the correct value of *result*, depending on the semantics of the specification. For instance if we chose the test set T_4, we can apply a regularity hypothesis of complexity 3 on the variable *p*, leading to the test set T_6:

T_6 = {<<cabin.create> <cabin.insert card> <cabin.enter first-pin> <cabin.buy *amount*> *T*, *result*>,
<<cabin.create> <cabin.insert card><cabin.enter new-pin first-pin> <cabin.buy *amount*> *T*, *result*>,
<<cabin.create><cabin.insert card><cabin.enter new-pin new-pin first-pin><cabin.buy *amount*> *T*, *result*> | *amount* ∈ money, *result* ∈ {*true*, *false*} s.t. the test is valid}

An uniformity hypothesis can be applied on this test for the variables *amount*, the new test T_7 includes now only ground term formulae.

T_7 = {<<cabin.create> <cabin.insert card> <cabin.enter first-pin> <cabin.buy 15> *T*, *result*>, <<cabin.create> <cabin.insert card> <cabin.enter new-pin first-pin> <cabin.buy 10> *T*, *result*>, <<cabin.create> <cabin.insert card> <cabin.enter new-pin new-pin first-pin> <cabin.buy 22> *T*, *result*> | *result* ∈ {*true*, *false*} s.t. the test is valid}

Then *result* can be evaluated in order to select a valid test set:

T_8 = {<<cabin.create> <cabin.insert card> <cabin.enter first-pin> <cabin.buy 15> *T*, *false*>, <<cabin.create> <cabin.insert card> <cabin.enter new-pin first-pin> <cabin.buy 10> *T*, *true*>, <<cabin.create> <cabin.insert card><cabin.enter new-pin new-pin first-pin> <cabin.buy 22> *T*, *false*>}

It must be noted that we have essentially tested in T_8 the correctness of the pin usage.

4 Conclusion

In this paper we have presented a generalization and an adaptation of the Bernot, Gaudel, and Marre theory of testing to object oriented software. The key points of the adaptation of the *BGM* method to the object oriented software is that we generalize this theory to systems where the specification and the test sets are given in a different language. Thus we are able to give some formal definitions of the test sets that can be used for systems having persistent states through the use of the *HML* temporal logic. It is also shown that these tests are fully compatible with the program satisfaction notions based on bisimulation. We illustrate our definitions through the Telephone example, a simple but sufficiently complete specification used to show the different kind of application of the reduction hypothesis on either object management mechanism, the event call part or the algebraic level of elements composing the *CO-OPN/2* specification.

Although our theory solve the most common problems of testing object-oriented software, it does not deal with some aspects of this paradigm, namely polymorphism and inheritance. Moreover, the presented strategy does not discuss the operational techniques for the reduction of the test set. Those aspects are not part of this paper and can be partly found in [2] and [1]. However, the test selection principles presented above have been designed to be incorporated as specific logical rules in *LOFT*. *LOFT* is an equational logic programming environment. Its purpose is to generate tests sets from algebraic specifications. We intend to extend this tool to handle object-oriented specifications, by coding the *HML* semantics rules and the *CO-OPN/2* class semantics into equational logic.

This paper proposes the first theoretical step in the direction of a complete practical methodology which will be enriched in the direction of concrete oracle construction, inheritance and operational techniques.

5 Acknowledgments

The authors would like to thank the anonymous referees and Pascal Racloz for their helpful comments. 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 (Office Fédéral pour l'Education et de la Science), and by the Swiss National Science Foundation project 21-36038.92 “Research in Object-Oriented Software Development".

6 References

[1] Stéphane Barbey, Manuel Ammann, and Alfred Strohmeier. Open issues in testing object-oriented software. In Karol Frühauf, editor, *ECSQ '94 (European Conference on Software Quality)*, pages 257– 267, Basel, Switzerland, October 17-20 1994. vdf Hochschulverlag AG an der ETH Zürich. Also available as Technical Report (EPFL-DI-LGL No 94/45).

[2] Stéphane Barbey, Didier Buchs, and Cécile Péraire. A theory of specification-based testing for object-oriented software. Technical Report 96/163, EPFL-DI-LGL, January 1996.

[3] Gilles Bernot, Marie-Claude Gaudel, and Bruno Marre. Software testing based on formal specifications: a theory and a tool. *IEE Software Engineering Journal*, 6(6):387–405, November 1991.

[4] Olivier Biberstein and Didier Buchs. Structured algebraic nets with object-orientation. In *Workshop on Object-Oriented Programming and Models of Concurrency '95*, Turin, June 1995.

[5] Didier Buchs and Nicolas 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.

[6] Roong-Ko Doong and Phyllis G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, April 1994.

[7] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, January 1985.

[8] Jarle Hulaas. An evolutive distributed algebraic petri nets simulator. In A. Javor, A. Lehmann, and I. Molnar, editors, *Modelling and Simulation 1996*, pages 348–352, Budapest, Hungary, June 2-6 1996. 10th European Simulation Multiconference ESM96, Society for Computer Simulation International.

[9] Bruno Marre. *Sélection automatique de jeux de tests à partir de spécifications algébriques en utilisant la programmation logique*. PhD thesis, LRI, Université de Paris XI, Orsay, France, January 1991.

[10] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[11] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[12] Elaine J. Weyuker. The oracle assumption of program testing. In *13th International Conference on System Sciences*, pages 44–49, Hawaii, USA, 1980.