Testing Against Formal Specifications:

a Theoretical View

Gilles Bernot
LIENS, URA CNRS 1327
45, rue d'Ulm
F-75230 Paris cedex 05
email: bernot@FRULM63.BITNET or bernot@dmi.ens.fr

1. Introduction

Assuming that a formal specification is available, one can formally study the validation of a software with respect to its specification. While proof theories are widely investigated, testing theories have not been extensively studied. The idea of deriving test data sets from a specification can be found in [Rig 85], [Scu 88], but there are few other published works. The work reported in this paper is the continuation of the works about formal specifications and testing reported in [Bou 85], [BCFG 85], [BCFG 86] and more recently [GM 88]. Some pioneering works on that subject were [GHM 81] and [GCG 85].

In practice, when big softwares are involved, a complete proof is often impossible, or at least it is not realistic because it would be too costly. The crucial properties of the program under test should be proved, but several less critical properties can be checked by testing. The aim of this paper is to propose a formal model of the testing approach. We assume that a formal specification (i.e. an axiomatic specification) is given and we have to validate a program against its specification: some of the axioms can be proved while the other ones can be tested.

Most of the current methods and tools for software testing are based on the structure of the program to be tested (''white-box'' testing). Using a formal specification, it becomes possible to also start from the specification to define some testing strategies in a rigorous and formal framework. These strategies provide a formalization of the well known ''black-box'' testing approaches. They have the interesting property to be independent of the program; thus, they result in test data sets which remain unchanged even when the program is modified. Moreover, such strategies allow to test if all cases mentioned in the specification are actually dealt with in the program.

The main advantage of our approach is to formally express what we do when testing. It allows also to modelize cases where some properties have been proved. Moreover, for algebraic specifications, we show that is is possible to automatically select test data sets from a structured specification. We have done a system developed in PROLOG, which allows to select a test data set from an algebraic specification and some hints about the chosen testing strategy. Our system is not described in this paper (it is described in [Mar 90] and [BGM 90]), but we define the strategies used by the system as examples of the theory.

The paper is organized as follows:

- Obviously, you are reading the introduction (Section 1) ...
- Section 2 contains an intuitive approach of our formalism
- Section 3 gives some preliminary notations and definitions
- Section 4 defines the fundamental notion of *testing contexts*
- Section 5 explains how testing contexts can be refined in order to produce ''practicable'' testing contexts
- Section 6 specializes to the case of *algebraic specifications*
- Section 7 shows our basic examples of testing strategies
- Section 8 explains how the problem of deciding success/failure of a test set is solved when the specification and the program are designed for testability
- Section 9 gives some hints on how to test less testable programs
- Section 10 recapitulates the main ideas of our theory.

2. Intuitive Approach

We introduce the important idea (already sketched in [BCFG 86]) that a *test data set* (i.e. a set of elementary tests) cannot be considered (or evaluated, or accepted, etc.) independently of:

- some *hypotheses* on the program which express the gap between the success of the test and the correctness of the program
- the existence of an *oracle*, i.e. a means of deciding, for any submitted test data, if the program behaves correctly with respect to its specification.

Thus, we define a *testing context* as a triple (H,T,O) where T is the test data set, H is a set of hypotheses and O is an oracle.
Then we state what is a *practicable* testing context, i.e. a context such that, assuming H,

O is able to decide the success or failure of the test data set T and T is successful via O if and only if the program is correct. We then define a "canonical" testing context which, unfortunately, is rarely practicable and we provide some way of *refining* it until we get a practicable testing context.

Intuitively, a program P gives a way of computing some operations and a specification SP states some properties which should be satisfied by these operations. For example, assuming that P is supposed to implement three operations named *sin*, *cos* and *tg*, an axiom of the specification SP could be:

$$\textbf{If} \quad x \equiv \frac{\pi}{2}\ [\pi] \quad \textbf{Then} \quad cos(x)=0 \quad \textbf{Else} \quad tg(x)=\frac{sin(x)}{cos(x)}$$

Running one test of this axiom consists of replacing the variable x by some constant α, computing by P the compositions of operations which occur in the axiom (i.e. $cos(\alpha)$ and $tg(\alpha)$, $sin(\alpha)$ if needed) and deciding, via the oracle, if the axiom is validated by P. It means that the oracle has to decide if ($\alpha \equiv \frac{\pi}{2}\ [\pi]$), it has to use the application rules of "**If..Then..Else**" (as defined by the semantics of the specification language of SP) and to decide if, for instance, $cos(\alpha)$ is equal to 0 (where "*is equal to*" is also defined according to the semantics of the specification language).

When a *test data set T* is defined as a set of ground instances of such axioms, our view of program testing is just a generalization of the classical way of running tests: the program is executed for a given input, and the result is accepted or rejected, according to the axioms which play the role of input-output relations required for the program.
Consequently, the correctness of these decisions is of first importance: if not ensured, it becomes possible to reject correct programs for instance. An oracle is some decision process which is able to decide, for each elementary test τ in T, if τ is successful or not when submitted to the program P. Thus, this decision is just a predicate, which will be denoted by O. Providing such an oracle is not trivial at all ([Wey 80], [Wey 82]) and may be impossible for some test data sets. Thus the existence of an oracle must be taken into account when selecting a test data set.

As usual, when a test data set is successful, the program correctness is not ensured, even if the oracle problem is solved: the set of all possible instances of the variables is generally infinite, or at least too large to be exhaustively experienced. However, when a test fails we know that the program is not correct: in some respects, it is consequently a success ...
Nevertheless, when the test is successful, we get a partial confidence in the program. We prefer to express this "partial confidence" as: "under what hypotheses does the success of the test data set imply the program correctness ?" These hypotheses are usually left implicit. We believe that it is of first importance to make them explicit. Moreover we consider that they are the good starting point for the selection of test data

sets: it seems sound to state first the hypotheses and then to select a test data set which ensures correctness, assuming these hypotheses.

Thus, given a formal specification *SP* and a program *P*, the test data selection problem implies to state some hypotheses *H* and to select some test data set *T* such that, informally:

$$H + success(T) \iff correctness(P,SP)$$

This equivalence can be shown as three implications:

- The *conservativity* means that the chosen hypotheses are satisfied by every correct program:
  $$correctness \implies H$$
- The *unbias* property means that a correct program cannot be rejected
  $$correctness \implies success$$
- The *validity* property means that under the hypotheses *H*, if the test is successful then the program is correct:
  $$H + success \implies correctness$$
  Equivalently, it also means that under the hypotheses *H*, any incorrect program is discarded (similar to the completeness criteria of Goodenough and Gehrart [GG 75]):
  $$H + incorrectness \implies failure$$
  Another interesting view of the same implication is the following
  $$incorrectness + success \implies \neg H$$
  which recalls that hypotheses are only ... hypotheses. Intuitively, there is a negative correlation between the strength of the hypotheses and the size of the selected test data set. If the selected test data set does not reveal any error, the hypotheses may be too strong.
  Notice that even when the hypothesis *H* is conservative, if the program *P* does not validate *H*, then the implication $H{+}success \implies correctness$ is always true (!)[*]. However, one should not be confused. In this case, the success of the test does not imply correctness because $(A \text{ and } B) \implies C$ is not equivalent to $B \implies C$, specially when *A* is false ... the implication is simply meaningless: the hypotheses are too strong.

Summing up this section, we have shown that it is not sufficient to simply select a test data set *T*. One should provide a related set of hypotheses *H* and a well defined oracle *O*, with the property:

$$H + \text{success of } T \text{ via } O \iff \text{correctness}$$

[*] I must thank one of the referees for this remark.

These three components are formally defined in Section 4. Let us first state several convenient notations.

3. Notations

We consider a rather flexible definition of "formal specification." It embeds various approaches of formal specifications such as algebraic specifications, temporal logic and other kinds of logic, etc. The readers familiar with Goguen and Burstall institutions will recognize a simplified version of them.

A formal specification method is given by a *syntax* and a *semantics.*

- The syntax is defined by a class of *signatures* and a set of *sentences* is associated to each signature.
  In practice, a signature Σ is a set of operation names. Given a signature Σ, the associated set of Σ-sentences, denoted by Φ_Σ, contains all the well-formed formulas built on: the operations in Σ, some variables, some atomic predicates and some logical connectives.
- The semantics is defined as follows: for each signature Σ there is an associated class of Σ-models, denoted by Mod_Σ, and there is a "validation predicate" on $Mod_\Sigma \times \Phi_\Sigma$, also called "the *satisfaction relation*", denoted by "$\models$". For each Σ-model $M \in Mod_\Sigma$ and for each Σ-sentence $\phi \in \Phi_\Sigma$, "$M \models \phi$" should be read as "M validates ϕ."

In this framework, a *formal specification* is a couple $SP=(\Sigma,Ax)$ such that Ax is a finite subset of Φ_Σ. The models of Mod_Σ which validate all the sentences of Ax are the models *validating* (or *satisfying*) SP. We denote by $Mod(SP)$ this class of models:

$$Mod(SP) = \{ M \in Mod_\Sigma \mid (\forall \phi \in Ax)(M \models \phi) \}$$

Of course the notions of signature, sentence, model and satisfaction relation depend on the kind of formal specification one need to consider. It is possible to verify the adequacy or inadequacy of a program P with respect to a specification SP only if the semantics of P and SP are expressible in some common framework. Thus the notion of model must be carefully defined.

When the signature Σ is just a set of operation names, a Σ-model M is usually a set of values and for each operation name of Σ, there is an operation of the relevant arity in M. Then, as a program gives a way of computing operations, one can consider that the behaviour of a program P defines a Σ_P-model, where Σ_P is the set of the names of all the operation exported by P.
It is important to note here that the variables, atomic predicates and logical connectives allowing to built the formulas of SP do not belong to the signature Σ. For instance SP

may contain existential quantifiers while P does not (and this is a usual case).
The model M_P associated with P is not well known *a priori*. It is the reason why validation techniques such as testing or proving must be used in order to check whether $M_P \in Mod(SP)$ (as ''P is correct'' is equivalent to ''$M_P \in Mod(SP)$'').

4. Testing Contexts

Definition :

Let P be the program under test. Let $SP=(\Sigma,Ax)$ be the specification that P is supposed to implement. A *testing context* is a triple (H,T,O) where:

- H is a set of hypotheses about the model M_P associated with P. This means that H describes a class of models $Mod(H)$. $Mod(H)$ is a subclass of Mod_{Σ_P} where Σ_P is the signature associated with P ($Mod(H)$ is called the class of models ''satisfying H'')
- T is a subset of Φ_Σ. It is called a ''test data set'' and each element τ of T is called an ''elementary test''
- the oracle O is a partial predicate on Φ_Σ. For each sentence (think ''each elementary test'') ϕ in Φ_Σ , either $O(\phi)$ is undefined either it decides if ϕ is successful when submitted to P.

This definition is very general and call for some comments:
The signature Σ_P associated with P contains, in practice, the names (and arities) of all the operations exported by P (as explained in Section 3 above). A priori we do not assume any adequacy between Σ_P and Σ (but we shall do it soon). Indeed, $Mod(H)$ will characterize a subclass of programs such that ''incorrectness'' implies ''failure of the test.''
It may seem surprising that test data sets can contain sentences with variables. Of course, our aim is to select test sets which are: executable (i.e. containing *ground* sentences), finite and mainly instances of the axioms of SP. However, the definition above is useful because it allows to build testing contexts by refinements.
The oracle O can be shown as a procedure using the program P: one should write O_P but for clarity of the notations P and SP are implicit parameters of the testing contexts.

Of course the goal of testing context refinements is to get so called practicable testing contexts:

Definitions :

Let (H,T,O) be a testing context.

1) (H,T,O) *has an oracle* means that:

   - T is finite (think "of reasonable size")
   - if M_P satisfies H then T is included in $D(O)$, where $D(O)$ is the definition domain of O (i.e. O is defined for each element of T)
   - if M_P satisfies H then O is decidable for each element τ of T.

2) (H,T,O) is *practicable* means that:

   - (H,T,O) has an oracle
   - if M_P satisfies H then: $O(T) \Leftrightarrow M_P \models Ax$
     where $O(T)$ denotes "$O(\tau)$ for all τ in T" which means that the test data set is successful via the oracle. Notice that $O(T)$ is defined and decidable when (H,T,O) has an oracle.

Let us describe some special examples of testing contexts.

The proof examples :

Let us assume that the syntaxes and semantics of P and SP allow theorem proving (e.g. P is a rewrite rule system and SP is an equational specification). One can consider the testing context where T is equal to Ax (the axioms of SP) and O is some theorem prover (e.g. by structural induction). One should notice that, even for this purely proof oriented example, the corresponding set of hypotheses H is not empty. In the example where P is a set of rewrite rules and O is based on structural induction methods, H must contain at least two hypotheses: the signature of P and SP are equal (i.e. $Mod(H) \subseteq Mod_\Sigma$) and every term submitted to P must be build on the operations of the signature (i.e. $Mod(H)$ only contains finitely generated models). Such hypotheses must not be neglected (more than 50% of the *proved* factorials admit negative arguments and loop on them !). The main advantage of our approach is that the hypotheses are *explicit*.
Such "proof oriented" testing contexts are practicable if and only if every axiom of SP is decidable via the theorem prover O.

The "lazy example" :

One can also consider an example where the hypothesis is "P is correct" (i.e. $Mod(H) = Mod(SP)$). Then T is empty and O is the undefined predicate. It reflects at least a full confidence in the program design. For instance, it can be used when the program has been automatically build from the specification, the hypothesis simply means that the program construction system is supposed correct. Most of the time, it seems sound to (try to) prove such hypotheses ...
Anyway, these testing contexts are always practicable.

The exhaustive test set :

Under the same set of hypotheses H as for the proof oriented case, one can consider the test data set T containing all ground instances of Ax. Assuming that there exists a decidable oracle for all ground instances, the resulting testing context is practicable if and only if T is finite (which is not the general case). Such an exhaustive test data set may work for enumerated types for instance.

Since the exhaustive test data set is generally infinite; since the empty test data set gives rise to an hypothesis which is too strong; since proving correctness is often too costly, it is clear that a good testing context is something in the middle of these three extremist views of testing. Such good and practicable testing contexts can be obtained via "testing contexts refinements." A natural starting point of these refinements is what we call the canonical testing context. It is build from the informations that we directly get from P and SP:

Definition :

The *canonical testing context* is defined by:

- the hypothesis "M_P is a Σ-model" which means $\Sigma_P=\Sigma$, i.e. $Mod(H) = Mod_\Sigma$
- the test data set $T = Ax$
- the oracle $O = undef$ (the never defined predicate)

Let us give some comments. It is not difficult to check in practice that the signature of P is Σ: it is just the set of exported operations implemented by P. Considering the axioms of SP as the canonical test data set simply means that our goal is to check whether P is compatible with the specification. Of course, the refinement process allows to select, step by step, a finite set of executable elementary tests. Similarly, the oracle, which is unknown at the starting point, is refined until it is decidable on the (refined) test data set. The crucial property of the canonical testing context is that it is valid (as proved in the next section).

Indeed, one need to define the validity and unbias properties (already sketched in Section 2): they provide sufficient conditions for practicability.

Definitions :

Let (H,T,O) be a testing context.

1) The test data set T is *valid* means that:
   If P satisfies H (i.e. $M_P \in Mod(H)$) then
   $$M_P \models T \quad \Longrightarrow \quad M_P \models Ax$$

2) The oracle O is *valid* means that:
   If $M_P \in Mod(H)$ then

$$(\forall \phi \in D(O))(O(\phi) \Rightarrow M_P \models \phi)$$

3) The test data set T is *unbiased* means that:
If $M_P \in Mod(H)$ then

$$M_P \models Ax \Rightarrow M_P \models T$$

4) The oracle O is *unbiased* means that:
If $M_P \in Mod(H)$ then

$$(\forall \phi \in D(O))(M_P \models \phi \Rightarrow O(\phi))$$

The sufficient condition for practicability is given by the following theorem:

Theorem :
Let (H,T,O) be a testing context. If (H,T,O) has an oracle and T and O are both valid and unbiased, then (H,T,O) is practicable.

The proof of this theorem is trivial. However, this theorem is fundamental since it justifies the testing refinement process: the canonical testing context defined above is valid and unbiased; in the following section we give a refinement criterion which preserves validity; and we give another criterion, for algebraic specifications, which ensures unbias. Consequently, practicability is ensured providing that we stop the refinement process when the triple (H,T,O) has an oracle (we shall explain how to get this result in a finite number of refinement steps).

5. The Refinement Preorder

Definition :
Let $TC_1 = (H_1,T_1,O_1)$ and $TC_2 = (H_2,T_2,O_2)$ be two testing contexts. TC_2 *refines* TC_1, denoted by $TC_1 \leq TC_2$, means that:

- $Mod(H_2) \subseteq Mod(H_1)$, (i.e. $H_2 \Rightarrow H_1$)
- $(\forall M_P \in Mod(H_2)) (M_P \models T_2 \Rightarrow M_P \models T_1)$
- if $M_P \in Mod(H_2)$ then $D(O_1) \subseteq D(O_2)$ and
$$(\forall \phi \in D(O_1)) (O_2(\phi) \Rightarrow O_1(\phi))$$

The first condition means that it is possible to encrease the hypotheses about the program under test via a refinement step. As already outlined in Section 2, encreasing the hypotheses allows to decrease the size of the selected test data set without loosing validity. This idea is reflected by the second condition. It exactly means that *under the new hypotheses* (H_2, which can be bigger than H_1), the new test data set T_2 must reveal an error if T_1 reveals an error. Examples where encreasing H allows to decrease T are given in Section 7. The third condition means that the oracle predicate can be build,

step by step, along the refinement preorder and that every error revealed by O_1 must be revealed by O_2.
Notice that the refinement defined above is clearly a preorder (reflexive and transitive); it is not antisymmetric.

The following theorem is trivial, but it is important because it ensures the validity of the test data set, providing that the refinement starts from the canonical testing context defined in Section 4 above:

Theorem :
Let (H,T,O) be a testing context. If (H,T,O) refines the canonical testing context then T is valid.

By the way, the refinement preorder allows trivially to express all the interesting properties defined in Section 4:

Propositions :
Let (H,T,O) be a testing context.
Let $Satisf_{D(O)}$ be the partial predicate on Φ_Σ defined by: the definition domain of $Satisf_{D(O)}$ is $D(O)$ and for each ϕ in $D(O)$, $Satisf_{D(O)}(\phi)$ is equivalent to $M_P \models \phi$. It simply means that $Satisf_{D(O)}$ coincides with the satisfaction relation "$\models$" in the definition domain of O.

1) The oracle O is valid if and only if $(H,T,Satisf_{D(O)}) \leq (H,T,O)$
2) The test data set T is unbiased if and only if $(H,T,O) \leq (H,Ax,O)$ (where Ax is the set of axioms of SP)
3) The oracle O is unbiased if and only if $(H,T,O) \leq (H,T,Satisf_{D(O)})$.
4) The set of hypotheses H is conservative if and only if $(H,T,O) \leq (Correct,T,O)$, where $Correct$ is the hypothesis defined by $Mod(Correct) = Mod_\Sigma(SP)$.

From **1)** and **3)** one can deduce that the oracle problem is to exhibit a decidable subdomain of the satisfaction relation which covers the selected test data set. Moreover, if the formal specification method under consideration has some Birkhoff's property, **2)** means that one must select elementary tests which are theorems of the axioms of SP.
In the rest of this paper, we show how to fulfill these criteria. For that purpose, we specialize to *algebraic specifications*.

6. Case of Algebraic Specifications

In this section, we first recall the main definitions of *algebraic abstract data types* as an instance of our general definition of formal specifications. Then we show that valid and unbiased test data sets can be selected from the so called "exhaustive test data set."

In the framework of algebraic specifications, a signature is: a finite set S of *sorts* (i.e. type-names) and a finite set of *operation-names* with arity is S. The corresponding class of models Mod_Σ is defined as follows: a Σ-model is a heterogeneous set M partitioned as $M = \{M_s\}_{s \in S}$, and with, for each operation-name "op: $s_1 \times \cdots \times s_n \rightarrow s$ " in Σ, a total function op^M: $M_{s_1} \times \cdots \times M_{s_n} \rightarrow M_s$.
The sentences of Φ_Σ are the positive conditional equations of the form:

$$(v_1{=}w_1 \wedge \cdots \wedge v_k{=}w_k) \implies v = w$$

where v_i, w_i, v and w are Σ-terms with variables ($k \geq 0$). A model M satisfies ($\models$) such a sentence if and only if for each substitution σ with range in M, if $\sigma(v_i){=}\sigma(w_i)$ for all i then $\sigma(v){=}\sigma(w)$ (as in [ADJ 76]).

Moreover specifications are structured: a *presentation* (or "specification module") $\Delta SP = (\Delta\Sigma, \Delta Ax)$ uses some *imported* specifications $SP_i = (\Sigma_i, Ax_i)$ such that $SP = \Delta SP$ + (*union of* SP_i) is a (bigger) specification. The signature Σ of SP is the disjoint union of $\Delta\Sigma$ and the union of the Σ_i; the set of axioms Ax of SP is the union of ΔAx and the Ax_i . For example, a *List* presentation (with a sorted insertion) over the imported specification of natural numbers can be expressed as follows:

ΔNATLIST uses *NAT* /* Here *ΔNAT* uses *BOOLEAN* */

$\Delta S = \{ \, NatList \, \}$

$\Delta\Sigma =$

$$\begin{array}{rlcl} empty: & & \rightarrow & NatList \\ cons: & Nat \times NatList & \rightarrow & NatList \\ ins: & Nat \times NatList & \rightarrow & NatList \end{array}$$

$\Delta Ax =$

$$\begin{array}{lcrcl} & & ins(n,empty) & = & cons(n,empty) \\ n{\leq}m = true & \implies & ins(n,cons(m,L)) & = & cons(n,cons(m,L)) \\ n{\leq}m = false & \implies & ins(n,cons(m,L)) & = & cons(m,ins(n,L)) \end{array}$$

where n and m are variables of sort *Nat*; L of sort *NatList.*

Of course, all the results stated in the previous sections remain for the particular case of structured algebraic specifications. In particular, given a testing context (H,T,O), an elementary test of T can be any positive conditional equation. The following definitions and results prove that it is possible to select ground instances of the axioms without loosing validity. Moreover, in that case, the unbias property is ensured.

Definitions :

1) A Σ-model M is *finitely generated* if and only if every value of M is denotable by a ground Σ-term. We denote by $Adequate_\Sigma$ the hypothesis such that $Mod(Adequate_\Sigma)$ is the class of finitely generated Σ-models.
This hypothesis means that the signature exported by P is exactly the signature of SP and that P does not admit inputs which are not denotable via Σ (we have already mentioned this hypothesis in the "proof example" given in Section 4).

2) Given a specification SP, the *exhaustive test data set*, $Exhaust_{SP}$, is the set of all ground instances of all the axioms of SP.
$$Exhaust_{SP} = \{ \sigma(\phi) \mid \phi \in Ax\ ,\ range(\sigma)=W_\Sigma \}$$
where W_Σ is the set of ground terms on Σ.

Under the Σ-adequacy hypothesis $Adequate_\Sigma$, every testing context which refines, and is included in the exhaustive test data set, produces valid and unbiased test data sets. More precisely:

Propositions :
Let (H,T,O) be a testing context.

1) If $T \subseteq Exhaust_{SP}$ then T is unbiased.

2) If (H,T,O) refines $(Adequate_\Sigma,Exhaust_{SP},undef)$ then T is valid.

Proposition 1) is trivial. Proposition 2) results from the fact that $(Adequate_\Sigma,Exhaust_{SP},undef)$ refines the canonical testing context defined in Section 4 and from the validity theorem of Section 5.

In particular the exhaustive test data set itself is unbiased and valid. Unfortunately it is generally infinite. The previous propositions means that the test data set refinement process can be reduced to "add hypotheses to $Adequate_\Sigma$ in order to select a subset of reasonable size from $Exhaust_{SP}$."
In the next section, we show via an example what kind of hypotheses can be used. Our system handles the corresponding refinements automatically.

7. Regularity and Uniformity

Let us assume that we want to test the axiom:
$$n \le m = false \quad \Rightarrow \quad ins(n,cons(m,L)) = cons(m,ins(n,L))$$
One has to select instances of the list variable L and instances of the natural variables m and n. A first idea, which is often used when testing, is to bound the size of the lists actually tested. It means to bound the size of the terms substituting L. This can be obtained via the general schema of *regularity hypothesis*.

Definition :

Let $\phi(L)$ be a sentence involving a variable L of a sort $s \in \Delta S$. Let $|t|_s$ be a complexity measure on the terms t of sort s (for instance the number of operations in $\Delta\Sigma$ of sort s occurring in t). Let k be a positive integer.

The *regularity hypothesis* of level k (with respect to ϕ and L), $Regul_{\phi(L),k}$, is the hypothesis which retains the Σ-models M_P such that:

$$(\forall t \in W_\Sigma)(\ |t|_s \leq k \Rightarrow M_P \models \phi(t)\ )\ \Rightarrow\ (\forall t \in W_\Sigma)(\ M_P \models \phi(t)\ )$$

where W_Σ is the set of all ground Σ-terms.

Instead of a complexity measure, any function such that the sets $\{t \in W_\Sigma \mid |t|_s \leq k\}$ are finite is acceptable. At first glance, it may seem unreasonable to use hypotheses which infer a result for every list by checking only small lists. Anyway, these hypotheses reflect what is done when testing. The main advantage of our approach is to make them *explicit*. Moreover one should never forget that the goal of testing is not to prove correctness, rather the goal of testing is to find good potential counter-examples of the correctness, as discussed in Section 2.

For the list example, a regularity level 3 allows to select the following instances of L:

$$\begin{array}{lcl} L & = & empty \\ L & = & cons(p,empty) \\ L & = & ins(p,empty) \\ L & = & cons(p,cons(q,empty)) \\ L & = & ins(p,cons(q,empty)) \\ L & = & cons(p,ins(q,empty)) \\ L & = & ins(p,ins(q,empty)) \end{array}$$

The corresponding refinement step of the testing context is obtained by adding the regularity hypothesis to H, and by replacing in T the axiom under test by the seven axioms related to those seven instances of L. Consequently, only variables of sort natural number remain: m, n, p and q.

Notice that the four instances of L involving the operation *ins* seems to be less relevant than those involving only *cons* and *empty* because *cons* and *empty* generate all the list values. Such a subset $\Delta\Omega$ of $\Delta\Sigma$ is called a set of generators. It is not difficult to modify the regularity hypothesis in order to get a so called Ω-regularity hypothesis which only retains the three instances of L build on *cons* and *empty* (as defined in [BGM 90]).

At this stage in the example, or after a finite number of regularity hypotheses in the general case, the test selection problem is reduced to the replacement of the variables of imported sort (natural number) by ground terms. The interesting cases correspond to the

4!=24 relative orders of m, n, p and q. These 24 arrangements provide 24 subdomains of N×N×N×N = N^4. A good testing strategy is to select one value in each subdomain. This can be obtained using *uniformity hypotheses* on these 24 subdomains.

Definition :

Let $\phi(V)$ be a formula involving a variable, or a vector of variables, V of imported sort(s). Let SD be a subdomain (i.e. a subset) of $W_{\Sigma,s_1} \times \cdots \times W_{\Sigma,s_k}$, where $s_1 \cdots s_k$ are the sorts of the variables of V. The *uniformity hypothesis* on the subdomain SD (with respect to ϕ and V), $Unif_{\phi(V),SD}$, is the hypothesis which retains the Σ-models M_P such that:

$$(\forall v_0 \in DS)(\ M_P \models \phi(v_0) \ \Rightarrow \ [\forall v \in DS][M_P \models \phi(v)]\ )$$

Such an hypothesis means that if a sentence is true for some value v_0 then it is always true in DS ... This is a strong hypothesis, and may seem unreasonable again. However, the same arguments hold: the hypotheses reflect what is done when testing; we have just made them *explicit*; and the goal is not to prove correctness, it is to find potential counter-examples of correctness.

A more interesting question is: "how to find the relevant subdomains ?". Our system is able to find them automatically. The key is: unfolding methods (based on equational logic programming with constraints). These methods are described in [Mar 90] or [BGM 90].

Notice that the modularity of the specification is crucial here: the regularity hypotheses are made for the sorts specified by ΔSP while the uniformity hypotheses are made for the imported sorts.

Very roughly, our system starts from a modular specification and translates the axioms into an equivalent set of Horn clauses ("equivalent" is defined in [Mar 90][BGM 90]). Then, for each axiom of ΔAx, the system selects a test data set as follows: for each sort of ΔS involved in the axiom, the system adds clauses reflecting the regularity hypotheses (the user gives the regularity level k); the axiom under test is transformed, via the regularity clauses, into a set of axioms where only variables of imported sorts remain; the unfolding methods are then applied to each transformed axiom, leading to a set of predicates (via the "wait" mechanisms) which define the uniformity subdomains; one instance of each subdomain is then selected (by resolution with a random choice of the clauses to apply).
Of course, many extensions of equational logic programming have been used, completeness being one of the main difficulties. The system is fully described in [Mar 90]; a simpler description can be found in [BGM 90].

8. The Oracle

In this section, we explain how the oracle problem can be solved using observability issues from the specification level.

Hypotheses such as those exposed in the previous section provide test selection strategies which allow to select finite test data sets included in $Exhaust_{SP}$. Thus, an elementary test is of the form:

$$(t_1=u_1 \wedge \cdots \wedge t_k=u_k) \Rightarrow t = u$$

where t_i, u_i, t and u are ground Σ-terms.
The oracle problem is reduced to decide success/failure of equalities between ground terms, because the truth tables of $\wedge$ and $\Rightarrow$ can then be used to decide success/failure of the whole conditional elementary test.

It is well known that deciding abstract equalities is not trivial at all. For example, let P implement stacks by means of arrays: *push* records the element at range *height* into the array and increments *height*; *pop* simply decreases *height*. Suppose the oracle has to decide if *pop(push(3,empty))* and *empty* give the same stack after their executions via P. There is an observability problem which, indeed, is a concrete equality problem: these stacks give two distinct array representations (because *3* has been recorded into the array for the first stack) but they are abstractly equal (as the common height is 0, and there is no observable access, via *top*, which allows to distinguish them). Thus, the predicate *O(pop(push(3,empty))=empty)* must be decided via some carefully elaborated method.

This problem relies on observability issues. Given a structured specification SP (e.g. *Stack*), a subset S_{Obs} of *observable sorts* is distinguished among the imported sorts (e.g. the natural numbers and the elements of the stacks). These observable sorts are specified by imported *observable specifications* SP_{Obs1}, SP_{Obs2}, etc. Intuitively, they correspond to the observable primitive types of the used programming language. In particular, for every program P, there are built-in equality predicates which are correct and defined on observable sorts.

There are mainly three approaches to handle the oracle problem, which are discussed in the rest of this paper.

A first idea is to modify the program under test in order to add new procedures extracting the concrete representations and computing the abstract equalities for non observable sorts. However, obviously, some of the advantages of the black-box approach are lost and the added procedures may alter the program behaviour. Then, the actually tested behaviour could not be represented by M_P any more, which results in a biased application of our formalism.

Indeed the proper approach is to take into account, from the specification stage, all these equality predicates for non observable sorts. This correspond to the well known slogan "plane *testable* programs at every stage of the design process." In this case, the oracle equality predicates are operations of Σ and our formalism can be directly applied.

However, since the oracle equalities are crucial for the quality of the performed tests, any hypothesis on them is very strong. Consequently, we recommand to put the sentences related to those equality predicates in the "proof part" of the testing context (Section 4). Then, the hypotheses (such as regularity and uniformity) are made only on the other sentences of the specification. This is "the good solution" for the oracle. However, when the program has not been designed to be easily testable, some partial results can be obtained via the second and third approaches which are discussed in the next section.

9. Partial Oracles

The second approach is to select only observable elementary tests. This can be done by replacing each non observable test $[t=u]$ by a set of terms of the form $[C(t)=C(u)]$ obtained by surrounding t and u with some well chosen *observable contexts C*. Here, the word "context" must be understood as "term with exactly one variable."
For example, an equality between two stacks $[t=u]$ seems to be equivalent to the following observable equalities: $[height(t)=height(u)]$, $[top(t)=top(u)]$, $[top(pop(t))=top(pop(u))]$ ... $[top(pop^{height(t)-1}(t))=top(pop^{height(t)-1}(u))]$. Here the chosen observable contexts C are $height(x)$ and the $top(pop^i(x))$ such that $i<height(t)$. Unfortunately, it is well known that identifying such a minimal set of contexts is undecidable for the general case of algebraic specifications [Sch 86]. Besides, when testing "big" stacks, this leads to an impracticable number of observable tests.

But the situation is even worst: when we think that $height(x)$ and $top(pop^i(x))$ are sufficient, we implicitly assume that we manipulate (more or less) stacks. But this fact is just what we check when testing !

Counter-example :

We sketch a program which does not satisfy this implicit hypothesis. The "bug" is that *top* returns the height when applied to a term t of the particular form $t=push(e,pop(...))$; for every other term t, $top(t)$ returns the correct value.
A *stack* is implemented by a record *<array,height,foo>* where *<array,height>* is the usual correct implementation and *foo* records the number of *push* performed since the last *pop* (and the empty stack is initialized with *foo*=2). We ignore the exceptional cases of *pop* and *top* when the stack is empty, it is not our purpose here ...

```
proc emptystack();
        stack.height := 0 ; stack.foo := 2 ;;

proc push(e:element);
        stack.array[stack.height] := e ;
        stack.height := stack.height+1 ; stack.foo := stack.foo+1 ;;

proc pop();
        stack.height := stack.height–1 ; stack.foo := 0 ;;

proc top();
        if (stack.foo = 1) then return stack.height   /* the bug */
        else return stack.array[stack.height] ;;
```

The terms *t=push(1,emptystack)* and *u=pop(push(2,push(1,emptystack)))* are distinguishable because *top(push(0,t))=0* (as *foo*=4) and *top(push(0,u))=2* (as *foo*=1). Nevertheless, the contexts *height(x)* and $top(pop^i(x))$ are unable to distinguish t from u (as *foo* is never equal to 1 for those contexts), leading to an oracle which never detects that $t \neq u$.

So, we get the depressing result that identifying the minimal set of observable contexts for the equality decision *is decidable* ! We must consider the set of *all* observable contexts ... which is infinite, consequently impracticable.
We can follow a similar method as for test selection: this infinite set of observable contexts can be reduced to a finite one by adding oracle hypotheses in the component H of the testing context. For example, the finite set of contexts mentioned for stacks is related to the following hypothesis on M_P:
"for all stacks t and u, if $M_P \models (height(t)=height(u))$ and $M_P \models (top(pop^i(t))=top(pop^i(u)))$ for $i<height(t)$ then $M_P \models (t=u)$."
It will be useful later to remark here that the number of observable contexts of the form $top(pop^i(x))$ is decided from the specification before the test is performed. Thus $i<height(t)$ should be understood with respect to the specification, $height(t)$ is not computed by the program here. Of course, the counter-example above does not meet this hypothesis and it will not be discarded by the selected observable test data set. However, one time more: the main advantage of our approach is to make the testing hypotheses *explicit*.

This idea of adding oracle hypotheses works when deciding equalities which appear *in the conclusion* of the tested conditional sentences. Unfortunately, if the equality appears in the precondition, for instance:

$$t = u \quad \Longrightarrow \quad concl\ ,$$

then we get a biased oracle ! This results from: $t=u$ may be successful according to the oracle hypotheses, but not satisfied in M_P. In that case, one would require for *concl* to be satisfied, in spite of the fact that *concl* is not required according to the formal satisfaction relation. Of course a first case where the solution is trivial is when the specification only contains sentences with observable preconditions (it is not really restrictive in practice). Another case which allows to solve this problem is when the specification is "complete" with respect to the imported observable specifications: if the oracle hypotheses are conservative then the oracle remain unbiased. All these techniques are more precisely studied in [Ber 89].

The third approach, somehow intermediate between the two first approaches, is an oracle which "drives" the program under test. The oracle is then realized as a new module on the top of the program under test. The new operations of this module are equalities for the non observable sorts where an equality predicate does not already exist in Σ_P. Notice that these oracle equalities are not "added into the code of P," rather they are computed on the top of P, and they can only exploit the observable results of P. For our stack example, we may add an equality predicate for the sort stack on the top of P as follows:

```
proc eq(t,u:stack);
    If (height(t)≠height(u)) then return false
    Else
        If (height(t)=0) then return true
        else return ( (top(t)=top(u)) and eq(pop(t),pop(u)) )   ;;
```

By convention, the terms which are computed by P have been written in **bold**.

Of course, this oracle is not able to discard the counter-example given above. This means that this oracle is valid (as defined in Section 4) under an additional oracle hypothesis. This hypothesis is similar to the hypothesis expressed for the second approach, except that the number of observable contexts of the form $top(pop^i(x))$ is dynamically computed by P: it depends on **height(t)** as computed by P; it does not depend on *height(t)* as specified by *SP*.
The difficult point here is that one must *prove* that the computed oracle equalities are sound with respect to the oracle hypotheses. If possible, we recommand to define them via rewrite rules because rewrite rules facilitate proofs in the framework of algebraic specifications. This approach is more detailed in [Ber 89].

Anyway, even if the approaches sketched in this section have some theoretical interest, practical experiments showed that they are not realistic: observable contexts (resp. oracle equalities) are often complex and there is a big number of them (for specifications of real size). This leads to an amount of potential errors which is almost comparable to the amount of faults in the program under test. It is far more suitable to plane testable programs from the specification stage, as recommended in Section 8.

10. Recapitulation

This paper gives a formal study of testing when a formal specification of the program under test is available. The main advantage is to make explicit, in a well defined framework, the limitations and the abilities of testing.

We have shown that a *test data set* cannot be considered, or evaluated, independently of: some *hypotheses* which express the gap between the success of the test data set and the correctness of the program; and the existence of an *oracle* which is a means of deciding success/failure of the test set. Thus, we have defined the fundamental notion of *testing context* which reflects theses three components.

Our definition of testing context is powerful enough to reflect a software validation approach where some of the properties required by the specification are proved, while the other ones are tested. This allows to mix proof methods and testing methods in a unified framework.

We have described, and formally justified, a refinement method which allow to get so called *practicable* testing contexts. Practicable testing contexts have finite test data sets and decidable oracles, and they have the following crucial property:

$$\text{Hypotheses + success of test via oracle} \iff \text{correctness}$$

When *algebraic specifications* are involved, we have established several more precise results which lead to a concrete way of building practicable testing contexts by refinements.

Several other works based on the theory described in this paper have been done, or are investigated:
We have done a system, based on equational logic programming with constraints, which automatically selects a practicable test data set from the specification and some hints about the corresponding hypotheses. Its existence proves that our formalism is usable.
We also extend our formalism to the case of algebraic specification with exception handling. It shall allow, in particular, to handle bounded data structures where performing tests "near the bounds" is crucial.

Acknowledgements :
Obviously, the work reported here is only the "basic theoretical part" of common researches about testing via formal specifications, and it is a pleasure to acknowledge *Marie Claude Gaudel*, *Bruno Marre* and *Pascale Le Gall* for numerous interactions and discussions which have allowed to build this formalism.
This work has been partially supported by the Meteor Esprit Project and the PRC "Programmation et Outils pour l'Intelligence Artificielle."

11. References

[ADJ 76] **J. Goguen, J. Thatcher, E. Wagner :** *An initial algebra approach to the specification, correctness, and implementation of abstract data types*, Current Trends in Programming Methodology, Vol.4, Yeh Ed. Prentice Hall, 1978.

[BCFG 85] **L. Bougé, N. Choquet, L. Fribourg, M. C. Gaudel :** *Application of PROLOG to test sets generation from algebraic specifications*, Proc. International Joint Conference on Theory and Practice of Software Development (TAPSOFT), Berlin (R.F.A), Springer-Verlag LNCS 186, pp.246-260, March 1985.

[BCFG 86] **L. Bougé, N. Choquet, L. Fribourg, M. C. Gaudel :** *Test sets generation from algebraic specifications using logic programming*, Journal of Systems and Software Vol 6, n°4, pp.343-360, November 1986.

[BGM 90] **G. Bernot, M. C. Gaudel, B. Marre :** *Software testing based on formal specifications: a theory and a tool*, To appear in Software Engineering Journal, U.K., 1991. (also: Internal Report LRI n°581, Orsay, France, June 1990.)

[Ber 89] **G. Bernot :** *A formalism for test with oracle based on algebraic specifications*, LIENS Report 89-4, LIENS/DMI, Ecole Normale Supérieure, Paris, France, May 1989.

[Bou 85] **L. Bougé :** *A proposition for a theory of testing: an abstract approach to the testing process*, Theoretical Computer Science 37, North-Holland, 1985.

[GCG 85] **C. P. Gerrard, D. Coleman, R. Gallimore :** *Formal Specification and Design Time Testing*, Software Science ltd, technical report, June 1985.

[GG 75] **J. B. Goodenough, S. L. Gerhart :** *Towards a theory of test data selection*, IEEE trans. soft. Eng. SE-1, 2, 1975. (Also: SIGPLAN Notices 10 (6), 1975.)

[GHM 81] **J. Gannon, P. McMullin, R. Hamlet :** *Data-Abstraction Implementation, Specification, and Testing*, ACM transactions on Programming Languages and Systems, Vol 3, no 3, July 1981, pp.211-223.

[GM 88] **M. C. Gaudel, B. Marre :** *Algebraic specifications and software testing: theory and application*, Internal Report LRI 407, Orsay, France, February 1988, and extended abstract in Proc. workshop on Software Testing, Banff, IEEE-ACM, July 1988.

[Mar 90] **B. Marre :** *Sélection automatique de jeux de tests a partir de specifications algebriques, en utilisant la programmation logique*, Ph. D. Thesis, LRI, Université de Paris XI, Orsay, France, January 1990.

[Rig 85] **G. Rigal :** *Generating Acceptance Tests from SADT/SPECIF*, IGL technical report, August 1986.

[Sch 86] **O. Schoett :** *Data abstraction and the correctness of modular programming*, Ph. D. Thesis, Univ. of Edinburgh, 1986.

[Scu 88] **G. T. Scullard :** *Test Case Selection using VDM*, VDM'88, Dublin, 1988, LNCS no 328 pp 178-186.

[Wey 80] **E. J. Weyuker :** *The oracle assumption of program testing*, Proc. 13th Hawaii Intl. Conf. Syst. Sciences 1, pp.44-49, 1980.

[Wey 82] **E. J. Weyuker :** *On testing non testable programs*, The Computer Journal 25, 4, pp.465-470, 1982.