Software Testing based on Formal Specifications: a theory and a tool

Gilles Bernot**
bernot@frulm63
bernot@dmi.ens.fr

Marie Claude Gaudel*
mcg@frlri61
mcg@lri.lri.fr

Bruno Marre*
marre@frlri61
marre@lri.lri.fr

Abstract

This paper addresses the problem of constructing test data sets from formal specifications.
Starting from a notion of an ideal exhaustive test data set which is derived from the notion of satisfaction of the formal specification, it is shown how to select by refinements a *practicable* test set, i.e. computable, not rejecting correct programs (*unbiased*), and accepting only correct programs (*valid*), assuming some hypotheses.
The hypotheses play an important role: they formalize common test practices and they express the gap between the success of the test and correctness ; the size of the test set depends on the strength of the hypotheses.

The paper shows an application of this theory in the case of algebraic specifications and presents the actual procedures used to mechanically produce such test sets, using Horn clause logic. These procedures are embedded in an interactive system which, given some general hypotheses schemes and an algebraic specification, produces a test set and the corresponding hypotheses.

Key Words: software testing, algebraic specifications, logic programming.

* **LRI**, UA CNRS 410, Université PARIS-SUD, F-91405 Orsay cedex

** **LIENS**, URA CNRS 1327, Ecole Normale Supérieure, 45 rue d'Ulm, F-75230 Paris cedex 05

Published in :

Software Engineering Journal, Vol. 6, Num. 6, November 1991, pp. 387-405.

Introduction

Most the current methods and tools for software testing are based on the program to be tested: they use some coverage criteria of the structure of the program (control flow graph, data flow graph, call graph etc.). With the emergence of formal specification languages, 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 of being independent of the program; thus they result in test data sets which remain unchanged even when the program is modified. This is specially important for regression testing. Moreover, such strategies allow to test if all the cases mentioned in the specification are actually dealt with in the program. It is now generally agreed that "black-box" testing and "white-box" testing are complementary.

This paper presents both a theoretical framework for testing programs against formal specifications and a system, developed in PROLOG, which allows to select a test data set from an algebraic specification and a testing strategy. All the "good" selection strategies, as defined in the theory, are supported by the system.

The starting point of the theoretical framework is the notion of an *ideal exhaustive test set* associated with the formal specification. The success of a program against this exhaustive test set is stated to be a correctness reference (under some hypotheses to be discussed later). Clearly, the definition of the exhaustive test set is determined by the semantics of the kind of formal specification which is used.

This exhaustive test set is of course not usable in practice since it is (most of the time) infinite and, moreover, not always decidable: there are some cases where it is not possible to decide whether or not an execution returns a correct result, mainly for reasons of observability. It is an aspect of the so called *oracle problem*. However, as said above, the exhaustive test set provides a theoretical correctness reference, and we show how to *select*, by successive refinements, test sets which are finite, decidable and keep some other good properties of the exhaustive test set.

The crucial notion in the selection process is the concept of *testing hypotheses*, already introduced in [3] for different purposes. Hypotheses represent and formalize common test practices. Very roughly, when one chooses a finite test set $T = \{\tau_1, \ldots, \tau_n\}$ satisfying some criteria to test a property $P(x)$, one assumes at least the following hypothesis:

$$[P(\tau_1) \land P(\tau_2) \land \ldots \land P(\tau_n)] \implies \forall x, P(x)$$

As a matter of fact, such an hypothesis is the result of the combination of several simpler and more sensible hypotheses. These hypotheses are usually left implicit. We believe that it is of first importance to make them explicit. Besides, we claim that they are a good conceptual tool for the selection of test data sets: it seems sound to state first some general hypotheses and then to select a test set corresponding to them. Actually, we propose to build these hypotheses by combination and specialization of some general hypothesis schemes.

Generally, stronger hypotheses will result in smaller test sets: the weakest hypotheses are the ones associated with the exhaustive test data set; the strongest one is that the

program under test is correct and it corresponds to ... an empty test set. The practically interesting pairs of hypotheses and test sets are obviously somewhere between these extremist views of testing. As usual in testing, the problem is to find a sound trade-off between cost and quality considerations.

As soon as specifications are handled in a formal framework and the testing strategies are expressed as hypotheses (i.e. formulas), one can consider the possibility of using proof-oriented tools for the selection of test data sets. These tools depend on the kind of formal specification in use: in this paper we present a tool based on "Horn clause logic" which allows to deal with algebraic specifications and produces test sets corresponding to the combination of some general hypothesis schemes. Our tool is even more elaborated since it helps in the choice of hypotheses.

Let us come back to the *oracle* problem, i.e. how to decide whether or not a program execution returns a correct result. The solutions to this problem depend both on the kind of formal specification and of the program: a property required by the specification may not be observable using the program under test. Most the formal specification methods provide a way to express observability. In this case, the program is assumed to satisfy the observability requirements (for instance to decide correctly the equality of two integers: it is an *oracle hypothesis*), and, following the same approach as above, an *exhaustive observable test set* can be associated with the specification. Then, testing hypotheses are used to select a finite subset of it. We present here such a solution in the framework of algebraic specifications.

Summing up, the theoretical framework presented here introduces the important idea that a test data set cannot be considered (or evaluated, or accepted, etc) independently of some hypotheses and of an oracle. 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 an oracle. We then construct some basic testing contexts (roughly, those corresponding to the ideal exhaustive test sets mentioned above) and we provide some way for deriving from them *practicable* testing contexts. Informally, a testing context (H, T, O) is practicable if: T is finite; O is defined on all the test data in T; it rejects no correct programs (*unbias*); and, assuming the hypotheses H, it accepts only correct programs (*validity*).

The paper is organized as follows:

- Section 1 introduces the general concepts and some theoretical results for a large class of formal specifications.
- Section 2 specializes the results of Section 1 to the case of structured algebraic specifications. It is shown that observability issues are crucial for the oracle problem. Besides, some fundamental hypotheses such as *uniformity hypothesis* or *regularity hypothesis* are introduced.
- Section 3 discusses the way these hypotheses can be combined.
- Section 4 presents the basic techniques which are used in the system. They are mainly extensions of equational logic programming
- Section 5 describes the system and the way it makes it possible to implement the strategies corresponding to the practicable testing contexts defined in Section 2. Moreover, the treatment of an example is reported.

1 Formal specifications and testing

1.1 Testing a program against its formal specification

In this first part, we consider that a specification is a set of formulas, written using a fixed set of logical connectors, and some operation names of a signature Σ. A program, which is supposed to implement the specification, provides a way of computing (for instance a function, a procedure) for each operation name of the signature Σ. These rather flexible definitions embeds various approaches of formal specifications such as temporal logic, algebraic specifications, etc[1].

Let *SP* be a formal specification and *P* be a program. It is possible to verify (by testing or by proving) the adequacy or inadequacy of *P* with respect to *SP* if the semantics of *P* and *SP* are expressible in some common framework. This role is ensured here by the concept of an interpretation for a given signature: intuitively a Σ-interpretation is a set of values plus, for each name in the signature Σ, an operation of the relevant arity on these values. We consider that the semantics of *SP* is a class of Σ-interpretations, and that *P* defines a Σ-interpretation. Then, the question of the correctness of *P* with respect to *SP* becomes: does the Σ-interpretation defined by *P* belong to the class of the interpretations of *SP*?

More precisely, a formal specification method is given by a *syntax* and a *semantics*.

- The syntax is defined by a notion of *signature*. With each signature Σ is associated a set of *sentences* Φ_Σ. Φ_Σ contains all the well-formed formulas built on Σ, some variables, and some logical connectives.

- The semantics is defined by a class of Σ-*interpretations*, Int_Σ, and a *validation predicate* on $Int_\Sigma \times \Phi_\Sigma$ denoted by $\models$. For each Σ-interpretation A and for each formula ϕ, "$A \models \phi$" should be read as "A validates ϕ".

In this framework, a *formal specification* is a pair $SP = (\Sigma, Ax)$ such that Ax is a (finite) subset of Φ_Σ.

> **Notation:** the class of interpretations *validating* *SP* is called the class of *models* of *SP* and is denoted by $Mod(SP)$:
>
> $$Mod(SP) \;=\; \{\ A \in Int_\Sigma \mid A \models Ax\ \}$$

The notions of signature, sentence, interpretation, and validation depend on the kind of formal specification, for instance algebraic specifications, temporal logic,...

What does it mean to test a program *P* against a Σ-formula $\phi(X)$? As said above, $\phi(X)$ is a well-formed composition of logical connectives, operation names of Σ, and variables in X. Running a test of $\phi(X)$ consists of replacing the variables of X by some constants, computing by *P* the operations of Σ which occur in ϕ and checking that the results returned by *P* satisfy the property required by the connectives.
For instance, let f, g, h, a, b belong to Σ, let $\vee$ be a connective, x,y some variables; and $\phi(x, y)$ the following formula:

$$(\ f(x,y) \;=\; g(x)\ )\ \ \vee\ \ (\ f(x,y) \;=\; h(y)\ )$$

[1] The readers familiar with Goguen and Burstall institutions will recognize a simplified version of them.

Let us note f_P, g_P, ... the functions computed by P for f, g, ... A test data for the formula above is:

$$(f(a,b) = g(a)) \lor (f(a,b) = h(b))$$

Running this test consists of computing the three values $f_P(a_P, b_P)$, $g_P(a_P)$, $h_P(b_P)$ and checking that the first one is equal either to the second one, or to the third one.

This view of program testing is just a generalization of the classical way of running tests, where the program is executed for a given input, and the result is accepted or rejected: in this case, the formula is the input-output relation required for the program.

We call a *test data set* of a Σ-formula $\phi(X)$ a set of instances of $\phi(X)$. As usual, when a finite test data set of reasonable size is successful, the program correctness is not ensured and when a test is not successful, we know that the program is not correct. As indicated in the introduction, this fact is expressed in our framework by hypotheses:
given a formal specification *SP* and a program *P*, the test data selection problem consists of stating some hypotheses *H* and to select some test data set *T* such that:

$$H + Success(T) \implies Correctness(P, SP).$$

When we get $Success(T)$ with an incorrect program, it means that the program does not meet the hypotheses *H*. The implication above is similar to the completeness criteria of Goodenough and Gehrart [14]: for every incorrect program that meets the hypotheses *H*, the test set *T* must fail.

In the discussion above, the oracle problem is hidden behind the notation *Success*. The oracle is some decision process, formalized by the predicate *Success*, which should be able to decide, for each elementary test τ in *T* if τ is successful or not when submitted to the program *P*. Providing such an oracle is not trivial at all ([32][33]) and may be impossible for some test data sets. Thus, the existence of *Success* must be taken into account, as well as the hypotheses *H*, when selecting a test data set *T*.

1.2 Testing contexts

Definition: let *P* be the program under test and let $SP = (\Sigma, Ax)$ be its formal specification. A *testing context* is a triple *(H,T,Success)* where:

- *H* is a set of hypotheses about the interpretation A_P associated with *P*. (This means that *H* describes a subset $Mod_\Sigma(H)$ of Int_Σ: the set of Σ-interpretations "validating *H*")
- *T* is a subset of Φ_Σ; *T* is called a "test data set" and each element τ of *T* is called an "elementary test"
- the *Success* oracle is a partial predicate on Φ_Σ; for each formula (think "each elementary test") ϕ in Φ_Σ, either $Success(\phi)$ is undefined either it decides if ϕ is successful in A_P.

This definition is very general and calls for some comments.
Success can be shown as a procedure using the program *P*: one should write $Success_P$ (*P* and *SP* are implicit parameters of all the testing context).
It may seem surprising that test data sets can contain formulas with variables. Of course, our aim is to select test sets which are: executable (i.e. some set of ground formulas),

finite, and as we will see later, instances of the axioms of *SP*. However, the definition above is useful because it allows to build testing contexts by refinements. A good starting point of what we call the "testing elaboration process" is the triple *("A_P is a Σ-interpretation" , Ax_{SP} , undef)* where *undef* is the never defined predicate (as for test sets, more sensible oracles are built by refinements). This initial testing context means that one wants to test the axioms of the specification, the oracle is not defined at all, and the hypothesis *"A_P is a Σ-interpretation"* simply means that each required functionality has been implemented (*not necessarily correctly* implemented). This hypothesis is equivalent to $Mod_\Sigma(H) = Int_\Sigma$.

The goal is to refine this initial testing context in order to get a practicable testing context. In the rest of this subsection, we give a sufficient condition to obtain practicability. Let us first formally define what is practicability.

Definitions: let *(H,T,Success)* be a testing context. Let us note $\mathcal{D}(Success)$ the definition domain of *Success*.

1. *(H,T,Success) has an oracle* means that:
   - *T* is finite[2]
   - If A_P validates *H*, then *T* is included in $\mathcal{D}(Success)$
2. *(H,T,Success)* is *practicable* means that:
   - *(H,T,Success)* has an oracle
   - If A_P validates *H* then $Success(T) \iff A_P \models Ax$
     where *Success(T)* denotes "$Success(\tau)$ for all τ in T"[3].

We need to define some more properties on test data sets and oracles. These properties are sufficient conditions for practicability: "unbias" properties avoid rejection of correct programs; "validity" properties ensure that, assuming the hypothesis *H*, any incorrect program is discarded.

Definitions:

1. The test data set *T* is *valid* means that:
   If A_P validates *H* then
   $$A_P \models T \quad \Longrightarrow \quad A_P \models Ax_{SP}$$
2. The oracle *Success* is *valid* means that:
   If A_P validates *H* then
   $$\forall \phi \in \mathcal{D}(Success),\ Success(\phi) \quad \Longrightarrow \quad A_P \models \phi$$
3. The test data set *T* is *unbiased* means that:
   If A_P validates *H* then
   $$A_P \models Ax \quad \Longrightarrow \quad A_P \models T$$

[2] Actually, one should ask for *T* to be "of reasonable size"
[3] *Success(T)* is defined if *(H,T,Success)* has an oracle.

4. The oracle *Success* is *unbiased* means that:
   If A_P validates H then

$$\forall \phi \in \mathcal{D}(Success),\ A_P \models \phi \implies Success(\phi)$$

A test data set is unbiased if and only if it contains only formulas which are theorems provable from the axioms of the specification. Equivalently, this means that a testing context should never require properties about the program which can discard correct programs.

The sufficient condition for practicability of testing contexts is given by the following fundamental fact.

> **Fact:** let *(H,T,Success)* be a testing context. If *(H,T,Success)* has an oracle and T and *Success* are both valid and unbiased, then *(H,T,Success)* is practicable.

The proof is trivial. However, this fact is fundamental since it justifies the testing elaboration process: the initial testing context presented above is valid and unbiased; in the following subsection we give a refinement criterion which preserves validity; and in Section 2 we give another criterion, for algebraic specifications, which preserves unbias. Consequently, practicability is ensured providing that we stop the refinement process when the triple *(H,T,Success)* has an oracle.

1.3 The refinement preorder

In this subsection, we define the refinement preorder and we show how it makes it possible to build only valid testing contexts.

Definition: let $TC_1 = (H_1, T_1, Success_1)$ and $TC_2 = (H_2, T_2, Success_2)$ be two testing contexts. TC_2 *refines* TC_1, denoted by $TC_1 \leq TC_2$, means that:

- "the hypotheses about the program under test may increase"

$$Mod_\Sigma(H_2) \subseteq Mod_\Sigma(H_1)$$

$$\text{or equivalently} \quad H_2 \implies H_1$$

- "under the hypotheses H_2, T_2 reveals as many errors as T_1 does"

$$\forall A_P \in Mod_\Sigma(H_2),\ A_P \models T_2 \implies A_P \models T_1$$

- "under the hypotheses H_2, the oracle $Success_2$ is more defined than, and consistent with $Success_1$"

$$\text{If } A_P \text{ validates } H_2 \text{ then}$$
$$\mathcal{D}(Success_1) \subseteq \mathcal{D}(Success_2) \text{ and}$$
$$\forall \phi \in \mathcal{D}(Success_1),\ Success_2(\phi) \implies Success_1(\phi)$$

The refinement criterion is clearly a preorder[4] relation. Thus, it allows to built testing contexts incrementally (because of the transitivity). Let *(H,T,Success)* be a testing context; we have the following results.

[4]it is not anti-symmetric

Fact: T is valid if

$$(\text{“}A_P \textit{ is a } \Sigma\textit{-interpretation”}, Ax_{SP}, undef) \leq (H,T,Success)$$

(The proof results directly from the definitions.) This fact is important because *(“A_P is a Σ-interpretation”,Ax_{SP},undef)* is the starting point of our testing context refinement process; thus, the validity of T is always ensured.

Fact: *Success* is valid if

$$(\text{“}A_P \textit{ is a } \Sigma\textit{-interpretation”}, \emptyset, [A_P \models _]_{|\mathcal{D}(Success)}) \leq (H,T,Success)$$

where $[A_P \models _]_{|\mathcal{D}(Success)}$ is the restriction of the formal validation predicate to the definition domain of *Success*.

(The proof results directly from the definitions.) We show in section 2 how to built oracles which trivially meet this property.

In this framework, the oracle is a decision procedure of the validity of the statement: $A_P \models \phi$. In most the cases, there is only a subset of the Σ-formulas ϕ such that this statement is practically decidable (see Section 2.3). We call *observable* those formulas such that $A_P \models \phi$ is trivially decidable for every program P. The oracle problem is then reduced either to select only observable tests (in that case the unbias property is trivial) either to extend observability by modifying the program under test, i.e. by adding procedures to P. However, this last solution is not always safe since the extensions may give biased results. In this paper we only consider the first solution. There are other possible approaches, for instance to require any program under test to be “fully observable”, i.e. to provide sufficient functionalities for deciding all the properties in its specification. This is related to the general problem of *design for testability* which obviously requires further research.

2 Algebraic specifications and testing

In the rest of this paper, we consider a special case of formal specifications: the theory of *algebraic abstract data types.* Section 2.1 recalls the main definitions; Section 2.2 shows that valid and unbiased test data sets can be selected from the so-called "exhaustive" test data set; Section 2.3 explains how the oracle problem can be handled and Section 2.4 proves that it is sufficient to select elementary tests reduced to a simpler form, namely ground equalities. This justifies that our system generates such simpler elementary tests selected from the exhaustive test data set.

2.1 Algebraic specifications

In the framework of algebraic specifications, a signature Σ is: a finite set S of *sorts* (i.e. type-names) and a finite set of *operation*-names with arity in S. The corresponding class of algebras Alg_Σ is defined as follows: a Σ-algebra is a heterogeneous set A partitioned as $A = \{A_s\}_{s \in S}$, and with, for each operation-name "$op : s_1 \times ... \times s_n \rightarrow s$" in Σ, a total function $op^A : A_{s_1} \times ... \times A_{s_n} \rightarrow A_s$. Σ-algebras correspond to the Σ-interpretations of part 1. The formulas of Φ_Σ are positive conditional equations of the form:

$$(v_1 = w_1 \ \wedge ... \wedge \ v_k = w_k) \quad \Longrightarrow \quad v = w$$

where v_i, w_i, v and w are Σ-terms with variables ($k \geq 0$). An algebra A validates ($\models$) such a formula if and only if for each variable assignment σ with range in A, if $\sigma(v_i) = \sigma(w_i)$ for all i then $\sigma(v) = \sigma(w)$ (as in [1]).
Moreover, specifications are structured: a *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* specification over the imported specification of natural numbers can be expressed as follows:

$\Delta NATLIST$ uses NAT /* Here ΔNAT uses $BOOLEAN$ */

$\Delta S = \{\ NatList\ \}$
$S_{Obs} = \{\ Nat,\ Bool\ \}$

$\Delta\Sigma =$

$$\begin{array}{llcrcl} empty & : & & & \rightarrow & NatList \\ cons & : & Nat & * \ NatList & \rightarrow & NatList \\ sorted & : & & NatList & \rightarrow & Boolean \\ insert & : & Nat & * \ NatList & \rightarrow & NatList \end{array}$$

Generators: *empty, cons*

$\Delta Ax =$

$$sorted(empty) = true$$
$$sorted(cons(N1, empty)) = true$$
$$sorted(cons(N1, cons(N2, L))) = and(\leq(N1, N2), sorted(cons(N2, L)))$$

$$insert(N1, empty) = cons(N1, empty)$$

$$\leq(N1, N2) = true \implies insert(N1, cons(N2, L)) = cons(N1, cons(N2, L))$$
$$\leq(N1, N2) = false \implies insert(N1, cons(N2, L)) = cons(N2, insert(N1, L))$$

where $N1$ and $N2$ are variables of sort *Nat*, L of sort *NatList*.

Note that, given a structured specification *SP*, a subset S_{Obs} of *observable sorts* is distinguished among the sorts (*Nat* and *Bool* in the example). In our case, these observable sorts are specified by imported predefined specifications. They correspond to the observable primitive types of the used programming language; in particular for every program P, we assume that there are built-in equality predicates which are correct and defined on the observable sorts.
We also declare a set of *generators* included in $\Delta\Sigma$. This means that every element of an algebra validating the specification must be denotable by a ground term built on those generators. For example, every list must be denotable by a composition of *empty* and *cons*.
Lastly, let us remark that the *uses* mechanism is transitive: if A uses B and B uses C then A uses C. This transitive view of imports is not the most common one in programming languages, but it is the most common one in specification languages.

2.2 The exhaustive test set

Of course, all the results of Section 1 remain for the particular case of structured algebraic specifications. In particular, given a testing context *(H, T, Success)*, an elementary test of T can be any positive conditional equation. This subsection proves that it is possible to select only ground instances of them without loosing validity. Moreover, in that case, we show that the unbias property is easily obtained.

Definition: given a specification ΔSP, the *exhaustive* test data set, $Exhaust_{SP}$, is the set of all ground instances of all the axioms of ΔSP.

$$Exhaust_{SP} = \{\sigma(\phi) \mid \phi \in \Delta Ax \ , \ range(\sigma) = W_\Sigma\}$$

where W_Σ is the set of ground terms on Σ.

Our first result about $Exhaust_{SP}$ is that it allows to select test data sets which are never biased. Remember that it was difficult to practically ensure this property in the general case of formal specifications (Section 1).

Fact: let *(H, T, Success)* be a testing context.
If $T \subseteq Exhaust_{SP}$ then T is unbiased.

(This fact results directly from the definitions.)

The Σ-*adequacy hypothesis* defined below means that the program under test only exports the specified operations (i.e. there are no exported operations which are not specified). Under this non restrictive hypothesis $Exhaust_{SP}$ always produces valid test data sets via testing context refinements.

Definition: let A_P be the algebra associated with P.

$$Adequat_\Sigma(A_P) \iff A_P \textit{ is a finitely generated } \Sigma\textit{-algebra}$$

(a Σ-algebra A is finitely generated if every value of A is denotable by a ground Σ-term.)

Fact: let *(H,T,Success)* be a testing context.
If $(H, T, Success) \geq (Adequat_{\Sigma}, Exhaust_{SP}, undef)$ then T is valid.

This fact results from:

$$(\text{“}A_P \text{ is a } \Sigma\text{-algebra”}, Ax_{\Delta SP}, undef) \leq (Adequat_{\Sigma}, Exhaust_{SP}, undef)$$

and from the validity fact of Section 1.3.

In particular the exhaustive test data set is valid; but unfortunately it is generally infinite. The previous facts mean that the testing context refinement process can be reduced to “add hypotheses in order to select a finite (pertinent) subset of $Exhaust_{SP}$.”

Let us show on an example what kind of hypotheses can be used. Let us treat the axiom

$$\leq(N1, N2) = true \implies insert(N1, cons(N2, L)) = cons(N1, cons(N2, L))$$

One has to select instances of the list variable L and instances of the natural number variables $N1$ and $N2$. A first idea can be to bound the size of the list terms substituting L. This can be obtained via the general schema of *regularity hypothesis.*

Regularity hypothesis: let $\phi(L)$ be a formula 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 of sort $s \in \Delta\Sigma$ occurring in t). Let k be a positive integer. A regularity hypothesis of level k, $Regul_{\phi,k}(A_P)$, is expressed as follows:

$$(\forall t \in W_{\Sigma})(\ |t|_s \leq k \ \Rightarrow\ A_P \models \phi(t)\ ) \quad \implies \quad (\forall t \in W_{\Sigma})(\ A_P \models \phi(t)\ )$$

where W_{Σ} is the set of all ground Σ-terms.

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

$L = empty$
$L = cons(N3, empty)$
$L = insert(N3, empty)$
$L = cons(N4, cons(N3, empty))$
$L = insert(N4, cons(N3, empty))$
$L = cons(N4, insert(N3, empty))$
$L = insert(N4, insert(N3, empty))$

The corresponding refinement of the testing context is obtained by adding the regularity hypothesis of level 3 to H, and by replacing the previous axiom by the seven axioms related to those seven instances of L. Consequently, only variables of sort natural number remain: $N1, N2, N3$ and $N4$.
The four instances of L involving the operation *insert* seem to be less relevant than those involving only *cons* and *empty* because *cons* and *empty* generate all the list values. Let us note $\Delta\Omega$ the set of generators. The operations of $\Delta\Sigma - \Delta\Omega$, such as *insert*, are called *defined operations.* Here, the treated axiom is one of the three *defining axioms* of *insert* in *NATLIST.*
Instances of L involving only generators of the sort s of L can be selected via the so called Ω-regularity hypothesis.

Ω-Regularity hypothesis: let $W_{\Delta\Omega+\Sigma_1...+\Sigma_n}$ be the set of the Σ-terms which do not contain defined operations (Σ_i are the imported signatures). Let $|t|_s$ be a complexity measure defined on those terms of sort s. A Ω-regularity hypothesis of level k is expressed as follows:

$$(\forall t \in W_{\Delta\Omega+\Sigma_1...+\Sigma_n})(\ |t|_s \leq k \ \Rightarrow\ A_P \models \phi(t)\ ) \quad \Longrightarrow \quad (\forall t \in W_{\Sigma})(\ A_P \models \phi(t)\ )$$

Then, a regularity level 3 allows to select only the three interesting instances of L. In our system, we mainly use a Ω-regularity hypothesis for each sort of ΔS and the user can choose the level k.

At this stage in the example, the test selection problem is reduced to the replacement of the variables of sort natural number by ground terms. In the general case, after a finite number of regularity hypotheses, only variables belonging to imported sorts remain.
The replacement of these variables by ground terms can be obtained "by brute force" using *uniformity hypothesis.*

Uniformity hypothesis: let $\phi(V)$ be a formula involving a variable V of imported sort s. A uniformity hypothesis, $Unif_{\phi}(A_P)$, is expressed as follows:

$$(\forall v_0 \in W_{\Omega,s})(\ A_P \models \phi(v_0) \ \Longrightarrow\ [\forall v \in W_{\Sigma,s}][A_P \models \phi(v)]\ )$$

Such an hypothesis means that if a formula is true for some value v_0 then it is always true ... This is a strong hypothesis, and it may seem unreasonable at first glance. However, it states explicitly what is often assumed when testing a program.

Of course, a uniformity hypothesis should not be applied directly to conditional axioms because one has few chance to validate the precondition of the axiom. For example, the formula

$$\leq(N1, N2) = true \ \Longrightarrow\ insert(N1, cons(N2, empty)) = cons(N1, cons(N2, empty))$$

could becomes

$$\leq(2, 0) = true \ \Longrightarrow\ insert(2, cons(0, empty)) = cons(2, cons(0, empty))$$

which is not relevant since $\leq(2,0)$ is false. Thus, uniformity hypothesis must be used carefully. We often use uniformity on appropriate *subdomains.* We show in Section 3 how we automatically derive these uniformity subdomains.

The modularity of the specification is crucial here. The choice of the hypotheses is guided by the specification structure: the regularity hypotheses are made for the sorts specified by ΔSP while the uniformity hypotheses are made for the imported sorts.

2.3 The oracle

In this subsection, we show how the oracle problem can be handled using observability issues. Some more elaborated solutions are described in [4].

Assuming that a finite test data set T has been selected from $Exhaust_{SP}$, an elementary test is of the form:

$$(t_1 = u_1 \land ... \land t_k = u_k) \ \Longrightarrow\ t = u$$

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 $\Longrightarrow$ can then be used to decide success/failure of the whole conditional axiom.
As already pointed out in [3], such a decision is not always trivial. For example, let P implements stacks by arrays: *push* records the element at range *height* 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 get two distinct array representations (because *3* has been recorded in the array for the first stack) but they are abstractly equal (as the common height is *0*). Thus $Success(pop(push(3, empty)) = empty)$ must be decided via some carefully elaborated method.

We may add new procedures to the program, extracting the concrete representations and computing the abstract equalities. However the added procedures may alter the program behaviour, they must be proved or tested; moreover, if they are added into the program code, some of the advantages of black-box testing are lost. A better solution could be to replace each test *[t=u]* by a set of tests of the form *[C(t)=C(u)]* obtained by surrounding t and u with some well chosen *observable contexts C*.
For the stack example, assuming that integers and elements are observable, $t{=}u$ seems to be equivalent to the observable following equalities: $[height(t) = height(u)]$, $[top(t) = top(u)]$, $[top(pop(t)) = top(pop(u))]$ ... $[top(pop^{height-1(t)}(t)) = top(pop^{height-1(t)}(u))]$. Here the observable contexts are $height(_)$ and the $top(pop^i(_))$ such that $0 \leq i < height(t)$. Unfortunately identifying such a minimal set of contexts is undecidable in the general case (see [30], page 8). Besides, when testing "big" stacks, this leads to an impracticable number of observable tests.

But the situation is even worst: when we think that $height(_)$ and $top(pop^i(_))$ 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 here a program which does not validate this implicit hypothesis. The "bug" is that *top* returns the height when applied to a term t of the particular form *t=push(x,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 of interest here ...].

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

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

proc pop();
  if (stack.height > 0) then
      { stack.height := stack.height-1 ; stack.foo := 0 } ;;
```

```
proc top();
  if (stack.foo = 1) then return stack.height   /* the bug */
     else if (stack.height > 0) then
          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 $top(pop^i(_))$ 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 the only credible alternative is to consider the set of *all observable contexts...* which is infinite (consequently impracticable).

A first idea is to follow a similar approach as for test selection: we may add oracle hypotheses in order to reduce this infinite set of contexts to a finite one, SC, with informally:

Oracle hypotheses and success(SC) $\Longrightarrow$ success of the original elementary test

Remark: let us consider the following oracle hypothesis:
"for all stacks t and u, if $height(t)=height(u)$ and $top(pop^i(t)) = top(pop^i(u))$ for $i=0..height(t)-1$ then $t=u$"
This hypothesis makes it posssible to reduce the infinite set of all contexts to the finite usual one: $height(_)$ and $top(pop^i(_))$.
The advantage of our approach is that the oracle hypothesis is explicitly mentioned. Moreover, for "big" stacks, a more powerful hypothesis can allow to select only a subset of all these *top*s. In such cases, the counter-example given above does not meet the oracle hypothesis and our black-box approach does not reveal the bug. Notice that any complementary white-box testing would find it (see the introduction).

The idea of adding oracle hypotheses works when deciding equalities which appear *in the conclusion* of the tested conditional axioms. If the program under test does not meet the oracle hypothesis then the oracle may accept wrong results, but this permissivity is just expressed by the hypothesis. Thus the validity of the testing context is not lost.
Unfortunately, if the equality appears in the precondition, for instance

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

then one get a biased oracle! This is due to the fact that $t=u$ may be successful according to the oracle hypotheses, but not valid. In that case one would require for *concl* to be true, in spite of the fact that *concl* is not required according to the formal validation predicate.

Fortunately a solution exists when the preconditions of all the axioms of SP belong to observable sorts only (which is the case for most specifications). Let us recall (Section 2.1) that there is a subset $S_{Obs} \subseteq S$ with implemented built-in correct equality predicates, denoted by $\{eq_s\}_{s \in S_{Obs}}$.

We first define the "minimal hypothesis" which formalizes the properties of the observable sorts, then we define the exhaustive observable test data set Obs_{SP}. Finally, we show that this data set allows to produce valid and unbiased oracles, in a similar way as $Exhaust_{SP}$ does for test data set without oracle.

Definition: the "minimal hypothesis", $Mini(A_P)$, is the conjunction of $Adequat_{\Sigma}(A_P)$ and the following:
for any ground equality $t{=}u$, if the sort s of t and u belongs to S_{Obs} then

$$A_P \models t = u \iff A_P \models eq_s(t,u) \iff SP \vdash t = u$$

(where "$\vdash$" stands for equational reasoning and structural induction[5]) else

$$A_P \models t = u \iff A_P \models eq_s(C(t), C(u)) \text{ for all observable contexts } C$$

This minimal hypothesis reflects the classical notion of correctness up to *behavioural equivalence* (as defined in [20, 31] or [18] for instance).

In the rest of this paper we assume that SP contains only axioms of the form $(L \implies R)$ where L is observable (or empty). Thus, the exhaustive test data set $Exhaust_{SP}$ contains elementary tests of the form $(L \implies t = u)$ where L contains only ground equalities of observable sorts. Of course, the problem of bias mentioned before does not remain since the built-in observable equality predicates are correct (from *Mini*).

Definition: the *exhaustive observable test data set* is the set Obs_{SP} whose elements are the ground formulas $[L \implies C(t) = C(u)]$ such that $(L \implies t = u)$ is element of $Exhaust_{SP}$ and C is any observable context over the sort of $t{=}u$.

The following definition and results show that one can reach practicability by selecting a finite subset of Obs_{SP}. This gives a solution to our problem.

Definition: $Success_{Obs}$ is the oracle defined as follows: $Success_{Obs}$ coincides with the implemented built-in predicates eq_s for all ground equalities of observable sort $s \in S_{Obs}$; $Success_{Obs}$ uses the truth tables of $\wedge$ and $\implies$ in a straightforward manner for conditional axioms involving only observable ground equalities; and $Success_{Obs}$ is undefined otherwise.

Lemma: for any testing context of the form $(H, T, Success_{Obs})$ such that $H \implies Mini$, the oracle $Success_{Obs}$ is valid and unbiased.

This results from the facts that the truth tables of $\implies$ and $\wedge$ are correct and complete with respect to the validation predicate and that the hypothesis *Mini* implies that each eq_s is valid, unbiased and defined for all observable ground equalities.

Lemma:

$$(Adequat_{\Sigma}, Exhaust_{SP}, undef) \leq (Mini, Obs_{SP}, undef)$$

In particular Obs_{SP} is a valid test data set under the hypothesis *Mini*.

[5] Note that equational reasoning and structural induction is correct and complete for *ground* equations

(The proof results directly from the definitions)

Fact: if *(H,T,Success)* is a testing context such that:

- $(H, T, Success) \geq (Mini, Obs_{SP}, Success_{Obs})$
- $T \subseteq Obs_{SP}$ and T is finite
- $Success = Success_{Obs}$,

then *(H,T,Success)* is practicable.

Proof: from the last fact of Section 1.2 it is sufficient to show that *(H,T,Success)* has an oracle, T and *Success* are unbiased, T and *Success* are valid.
Since *Obs* involves only observable ground conditional equalities and *Mini* implies that eq_s is defined on every observable ground equality, the fact that H is stronger than *Mini* implies that the definition domain of $Success_{Obs}$ contains Obs_{SP}. Consequently since T is a finite subset of Obs_{SP}, $(H, T, Success_{Obs})$ has an oracle.
Since $Exhaust_{SP}$ is unbiased and Obs_{SP} is built from $Exhaust_{SP}$ by adding contexts, Obs_{SP} remain unbiased.
The validity of T results from the inequality given in the previous fact and from the first fact of Section 1.3.
Moreover $Success_{Obs}$ is unbiased and valid from the previous fact. This ends the proof.

From the theoretical point of view, the previous fact is fundamental because it gives useful sufficient conditions to solve the test selection and oracle problems. From now on, one can consider only elementary tests of the form $[L \implies R]$ where all the equalities occurring in L and R are observable (consequently decidable). Our last theoretical improvement is to show that one can skip the elementary tests such that L is not satisfied.

2.4 Reducing test to equalities

Let us consider a test of the form $[L \implies R]$ such that L contains only observable ground equalities and R is a ground equality. Let us assume that L is not satisfied by the program under test. From the truth table of $\implies$, $[false \implies R]$ is always true. It follows that the elementary test $[L \implies R]$ has no chance to reveal any error of the program; thus it is (at least intuitively) useless (see Section 1.1). Consequently it is of first interest to select only elementary tests of the form $[L \implies R]$ such that L is true. This means that, given a conditional axiom of ΔSP

$$(v_1 = w_1 \wedge ... \wedge v_k = w_k) \implies v = w \ ,$$

one should select instances of the variables which satisfy the precondition $(v_1 = w_1 \wedge ... \wedge v_k = w_k)$. It is the reason why an equational resolution procedure *a la PROLOG* is needed for automatizing test selection (see sections 4 and 5).
Now, let us assume that the precondition L is true. From the truth table of $\implies$, $[true \implies R]$ is always equivalent to R alone. It follows that submitting $[L \implies R]$ is useless, it is sufficient to submit R to the program under test. Consequently, assuming that the specification allows to decide whether the precondition is true or false, our system can select only instances of

$$(v_1 = w_1 \wedge ... \wedge v_k = w_k) \implies v = w \ ,$$

satisfying $(v_1 = w_1 \wedge ... \wedge v_k = w_k)$ and produce only the related instances of $v{=}w$.

The only difficulty is that "L is true with respect to the specification" is *a priori* not equivalent to "L is true with respect to the program." Indeed, the following definition and results prove that there is no problem.

> **Definition:** the *exhaustive equational test data set* is the set $EqExh_{SP}$ whose elements are the ground equalities *[t=u]* such that:
> $(t_1 = u_1 \wedge ... \wedge t_k = u_k \implies t = u)$ is element of $Exhaust_{SP}$ and each equality $t_i = u_i$ is provable from the specification (using equational reasoning and structural induction).

> **Fact:** let *(H,T,Success)* be a testing context. If $T \subseteq EqExh_{SP}$ then T is unbiased.

Proof: let us remind that if *T* only contains theorems of *SP* then it is unbiased (Section 1.2). By definition, $EqExh_{SP}$ only contains theorems of the axioms of *SP* (because equational reasoning and structural induction is correct), which implies the fact.

> **Fact:** let *(H,T,Success)* be a testing context. Let us assume that ΔSP contains only axioms whose preconditions are observable.
> If $(H, T, Success) \geq (Mini, EqExh_{SP}, undef)$ then T is valid.

Proof: from the validity fact of Section 2.2 and by transitivity of "$\geq$", it is sufficient to prove that:

$$(Mini, EqExh_{SP}, undef) \geq (Adequat_{\Sigma}, Exhaust_{SP}, undef)$$

Thus it is sufficient to prove:

$$\forall\ A_P \in Mod(Mini)\ ,\quad A_P \models EqExh_{SP} \implies A_P \models Exhaust_{SP}$$

Let us assume that A_P validates *Mini* and that $A_P \models EqExh_{SP}$. Let $(t_1 = u_1 \wedge ... \wedge t_k = u_k \implies t = u)$ be an elementary test τ of $Exhaust_{SP}$; one has to prove that $A_P \models \tau$. Since A_P validates *Mini* and $t_i = u_i$ is observable for every $i = 1..k$, two cases are possible:

- $A_P \models eq_{s_i}(t_i, u_i)$ for every $i = 1..k$. In that case, $[t = u]$ belongs to $EqExh_{SP}$ because $SP \vdash t_i = u_i$ from the *Mini* hypothesis. Consequently, since $A_P \models EqExh_{SP}$, $A_P \models t = u$, which implies that $A_P \models \tau$.
- there exists i such that A_P does not validate $eq_{s_i}(t_i, u_i)$. In that case, A_P does not validate $t_i = u_i$ (from the *Mini* hypothesis). Consequently, $A_P \models \tau$ (as the precondition of τ is not true).

The two previous facts prove that it is sufficient to submit equational elementary test instead of conditional ones. In fact it is possible to select only equational observable elementary tests, as stated below:

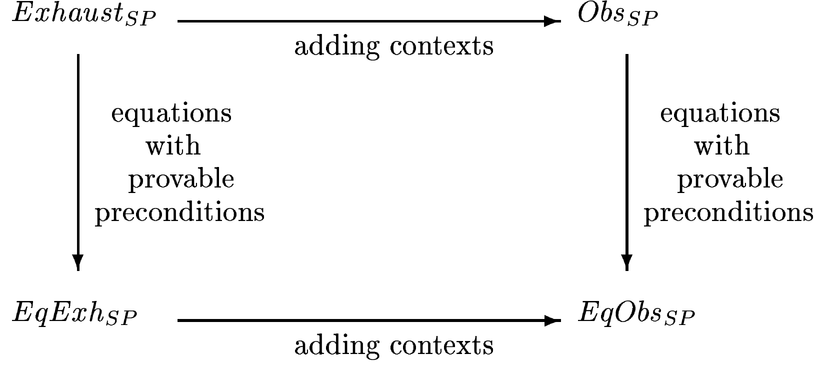
> **Definition:** the *exhaustive observable equational test data set* is the set $EqObs_{SP}$ whose elements are the ground equalities *[t=u]* such that:
> $(t_1 = u_1 \wedge ... \wedge t_k = u_k \implies t = u)$ is element of Obs_{SP} and each equality $t_i = u_i$ is a theorem of the specification (using equational reasoning and structural induction).

Fact: let us assume that ΔSP contains only axioms whose preconditions are observable. If *(H,T,Success)* is a testing context such that:

- $(H, T, Success) \geq (Mini, EqObs_{SP}, \{eq_s\}_{s \in S_{Obs}})$
- $T \subseteq EqObs_{SP}$ and T is finite
- $Success = \{eq_s\}_{s \in S_{Obs}}$

then *(H,T,Success)* is practicable.

Proof: $EqObs_{SP}$ can be also obtained from $EqExh_{SP}$ as follows: the elements of $EqObs_{SP}$ are the ground equations of the form $C(t) = C(u)$ such that $t = u$ is element of $EqExh_{SP}$.



The previous fact can be proved in a similar way as the fact of Section 2.3 above (by replacing $Exhaust_{SP}$ and Obs_{SP} by $EqExh_{SP}$ and $EqObs_{SP}$ respectively).

These results provide some guidelines for the selection of test data sets for a specification module ΔSP: the problem is to select a finite subset of $EqObs_{SP}$. As we already said, the reduction of $EqObs_{SP}$ to a finite subset is obtained by stating regularity (or Ω-regularity) hypotheses on the defined sorts, uniformity hypotheses on the imported sorts and oracle hypotheses for the observable contexts. The previous fact proves that the resulting testing contexts are practicable. However, we have seen that the composition of these hypotheses must be done carefully: we gave an example with a conditional axiom in Section 2.2. Indeed, we are faced to a more general problem which is discussed below.

3 Choice of regularity and uniformity hypotheses

3.1 The problem

Putting together uniformity hypotheses and regularity hypotheses is not straightforward. Brute uniformity hypotheses turn out to be too strong, in some cases for equations and in most the cases for conditional axioms. By too strong, we mean that some obviously relevant cases are not tested under these hypotheses.

An instance of the problem for a conditional axiom has been given in section 2.2: applying uniformity hypotheses on some sorts consists of assigning arbitrary values to the variables of these sorts ; this can result in an instance of the axiom where the precondition is false ; thus testing this instance is meaningless. In this case, the uniformity hypotheses must not be made on the whole domains of the variables, but on a subdomain which is the validity domain of the precondition.

The notion of uniformity subdomain is not only useful for conditional axioms, but also, even if it is less obvious, for equations. Let us consider the following equation of the specification of lists of natural numbers :

$$sorted(cons(N1, cons(N2, L))) = and(\ \leq(N1, N2), sorted(cons(N2, L)))$$

Ω-regularity hypothesis of level 2 on lists yields the following set of instances:

1. $sorted(cons(N1, cons(N2, empty)))$
   $= and(\ \leq(N1, N2), sorted(cons(N2, empty)))$

2. $sorted(cons(N1, cons(N2, cons(N3, empty))))$
   $= and(\ \leq(N1, N2), sorted(cons(N2, cons(N3, empty))))$

A uniformity hypothesis on natural numbers would result in two ground equations (the remaining *Nat* variables are replaced by some ground terms). It has good chance to result in a test data set where some interesting cases in the definition are not covered. For instance:

$$sorted(cons(5, cons(3, empty))) = and(\ \leq(5, 3), sorted(cons(3, empty)))$$
$$sorted(cons(4, cons(2, cons(6, empty)))) = and(\ \leq(4, 2), sorted(cons(2, cons(6, empty))))$$

Indeed, a better coverage should at least reach the following cases:

- $\leq(N1, N2) = true\ \wedge\ sorted(N2, L) = true$
- $\leq(N1, N2) = true\ \wedge\ sorted(N2, L) = false$
- $\leq(N1, N2) = false\ \wedge\ sorted(N2, L) = true$
- $\leq(N1, N2) = false\ \wedge\ sorted(N2, L) = false$

(In the test data set above, only the third case is covered). These four cases define four subdomains for the tested axiom. These subdomains come from the decomposition of the definition of *sorted* and from the properties of *and*. Uniformity hypotheses on these subdomains can be made; in that case, they are called *uniformity subdomains*. However, the decompositions could be continued using the axioms of $\leq$ or the defining axioms of *sorted* (including the one under test).

3.2 Decomposition of subdomains by unfolding defined operations

What we have done above is a decomposition of the domain of *sorted*. In this subsection we describe how to decompose defined operations in a systematic way, in order to get relevant uniformity subdomains for the axioms where they occur[6].

As usual in testing methods, this decomposition is based on case analysis: the preconditions which occur during the decomposition are composed by conjunction. This is done until a satisfactory coverage of the cases mentioned in the specification is obtained. Of course, the level of case coverage is strongly correlated to the size of the data set. It is not always possible or realistic to push the decomposition too far. At the end of this section, we give some hints on how to stop the decomposition with relevant associated uniformity hypotheses.

In order to describe the decomposition of defined operations in a legible way, we assume that the specification has the following form: for each defined operation f there is a set of defining axioms where f is the top symbol of the left hand side of the conclusion. Thus a defining axiom looks like:

$$precondition \Rightarrow f(t_1, \ldots, t_n) = t$$

where *precondition* may be either empty or a conjunction of equations. We assume that the left hand-side of the conclusion is defined by the right hand-side. Thus, axioms are implicitly oriented from left to right.

Let us assume that we want to find subdomains for a defining axiom of f. Let $\phi_f = (precond_f \Rightarrow concl_f)$ be this axiom. Suppose that there is an occurrence u of a defined operation g in ϕ_f (which is not the top of the left hand-side of $concl_f$). Let $\phi_g = (precond_g \Rightarrow g(t_1, \ldots, t_m) = t)$ be one of the axioms defining g. Let $g(a_1, \ldots, a_m)$ be the subterm at the occurrence u in ϕ_f. The unfolding of ϕ_f via ϕ_g at the occurrence u can be described in two steps:

- we first replace $g(a_1, \ldots, a_m)$ by t in ϕ_f at the occurrence u. Let $(precond' \Rightarrow concl')$ be the resulting formula.
- then we add into $precond'$ the equalities between the related arguments of g and the precondition of ϕ_g. Thus, the unfolded axiom is the following one:

$$(precond' \ \wedge \ t_1 = a_1 \ \wedge \ldots \wedge \ t_m = a_m \ \wedge \ precond_g \Rightarrow concl')$$

This unfolding is done at occurrence u with all axioms defining g, giving as many unfolded axioms as there are axioms defining g.

It appears that for testing ϕ_f we have the choice between:

- replacing directly its variables by ground terms which satisfy $precond_f$ (without unfolding), then the corresponding hypothesis is uniformity on the validity domain of $precond_f$

[6] The result of these decompositions is a partition of the domain of the axiom where the operations occur into subdomains. The original axiom, which is unchanged, will be tested once for each subdomain.

- performing a similar replacement for each of the unfolded axioms. Of course, the test data set is larger. We get more, but weaker, uniformity hypotheses.
- if there is still some occurrence of a defined operation in one of the unfolded axioms, continuing the subdomains decomposition by unfolding (before replacing).

When g is f itself (recursive definition of f), this transformation is similar to the classical Burstall and Darlington's unfolding of recursive definitions.

It is clear that in most the cases, such a decomposition is infinite. However, assuming some regularity and uniformity hypotheses (on subdomains), it is possible to make these decompositions finite, as seen in the example below.

Let us come back to our axiom on *sorted* (section 2.1), and assume that the defining axioms of *and* and $\leq$ are:

$$
\begin{array}{ll}
and(true, true) = true & \leq(N1, N1) = true \\
and(true, false) = false & <(N1, N2) = true \Rightarrow \leq(N1, N2) = true \\
and(false, true) = false & <(N2, N1) = true \Rightarrow \leq(N1, N2) = false \\
and(false, false) = false &
\end{array}
$$

where $<$ is inductively defined on the natural numbers generated, as usual, by zero and successor. For legibility purpose, these defining axioms explicitly reflects the cases that we want to cover in the decompositions.

As seen above, a regularity hypothesis of level 2 gives the two following instances:

1. $sorted(cons(N1, cons(N2, empty)))$
   $= and(\ \leq(N1, N2), sorted(cons(N2, empty)))$
2. $sorted(cons(N1, cons(N2, cons(N3, empty))))$
   $= and(\ \leq(N1, N2), sorted(cons(N2, cons(N3, empty))))$

Let us consider the first equation.

1. From the first axiom of $\leq$, it comes:

   $$
   \begin{array}{l}
   N1 = N2 \Rightarrow \\
   \quad sorted(cons(N1, cons(N2, empty))) = and(true, sorted(cons(N2, empty)))
   \end{array}
   $$

   From the definition of *and*, we get two cases:

   - the recursive call of *sorted* is true

     $$
     \begin{array}{l}
     N1 = N2 \ \wedge \ sorted(cons(N2, empty)) = true \Longrightarrow \\
     \quad sorted(cons(N1, cons(N2, empty))) = true
     \end{array}
     $$

     and from the axioms of *sorted* we get $true = true$ in the precondition, which can be suppressed (indeed, there are more cases, but each of them lead to an unsatisfiability in the precondition).
     A uniformity hypothesis on the domain of $N1 = N2$ seems sensible, thus we stop the decomposition.

- if the recursive call of *sorted* is false, we get $true = false$ in the precondition, thus we ignore this irrelevant case.

2. From the second axiom of $\leq$, it comes:

$$<(N1, N2) = true \Rightarrow \\ sorted(cons(N1, cons(N2, empty))) = and(true, sorted(cons(N2, empty)))$$

From the axioms of *and*, we get only one case (following the same method as in 1).

$$<(N1, N2) = true \ \wedge \ sorted(cons(N2, empty)) = true \Longrightarrow \\ sorted(cons(N1, cons(N2, empty))) = true$$

and from the axioms of *sorted* we can eliminate $sorted(cons(N2, empty)) = true$. A uniformity hypothesis on the domain of $<(N1, N2) = true$ seems sensible, thus we can stop the decomposition.

3. From the last axiom of $\leq$, it comes:

$$<(N2, N1) = true \Longrightarrow \\ sorted(cons(N1, cons(N2, empty))) = and(false, sorted(cons(N2, empty)))$$

From the axioms of *and*, we get two cases:

- the recursive call of *sorted* is true

$$<(N2, N1) = true \ \wedge \ sorted(cons(N2, empty)) = true \Longrightarrow \\ sorted(cons(N1, cons(N2, empty))) = false$$

and from the axioms of *sorted* we can eliminate the second equality of the precondition.
As in the previous case, a uniformity hypothesis on the domain of the precondition seems sensible, thus we stop the decomposition.

- if the recursive call of *sorted* is false, we get a contradiction with the axioms defining *sorted*, thus we ignore this case.

All the preconditions of these cases define the coverage we want for the first instance of the axiom. They define the uniformity subdomains of the domain of the axiom under test (in this case, the natural numbers, since the axiom is not conditional) as follows:

- $N1 = N2$
- $<(N1, N2) = true$
- $<(N2, N1) = true$

Thus, for the first instance of the axiom generated by regularity, we select three instances (one for each uniformity subdomain).

For the second instance of the axiom generated by regularity, we follow the same way and get nine decomposition subdomains defined by:

- $N1 = N2 \ \wedge \ N2 = N3$
- $N1 = N2 \ \wedge \ <(N2, N3) = true$

- $N1 = N2 \ \wedge \ <(N3, N2) = true$
- $<(N1, N2) = true \ \wedge \ N2 = N3$
- $<(N1, N2) = true \wedge \ <(N2, N3) = true$
- $<(N1, N2) = true \wedge \ <(N3, N2) = true$
- $<(N2, N1) = true \ \wedge \ N2 = N3$
- $<(N2, N1) = true \wedge \ <(N2, N3) = true$
- $<(N2, N1) = true \wedge \ <(N3, N2) = true$

Clearly, the selected uniformity subdomains depends both of the axioms of the specification and of the criteria we choose to stop the decompositions. This is not surprising since any black-box testing method depends on the specification. We present in the next section a tool which enables us to stop such decompositions using some control specifications. This tool provides a mean to automatically compute uniformity subdomains for each axiom.

4 Basic tools for automatizing test data selection

This part presents in a detailed way the principles and internal mechanisms of a system which makes it possible to automatize the approach presented in parts 2 and 3. This system takes as input an algebraic specification and some indications on regularity hypotheses and halting of unfolding. It provides uniformity subdomains and the corresponding test data. The reader which is not familiar with logic programming techniques can skip this part and go to part 5 where an example of the use of the system is given. Part 5 refers to part 4 but is understandable by itself.

Two of the main difficulties in automatizing the selection are:

1. the computation of the elements of a uniformity subdomain
2. the decomposition, using the axioms, of a domain into uniformity subdomains.

we present here a procedure and some control strategies which allow to cope with these difficulties.

To solve the first point, we need an equational resolution procedure to compute the solutions of the equations that define a uniformity subdomain, which of course must be correct (any returned value is a solution) and complete (all the solutions are computed). In the case of positive conditional axioms, there exist such procedures: the conditional narrowing presented in [19] for the RAP language; the clausal superposition for equational Horn clauses presented in [10] for the SLOG language.

In a first step, we performed several experiments for automatizing the test data selection using RAP and SLOG. These experiments allowed us to study and to identify the specific control mechanisms for the implementation of our selection strategies. The introduction of these control mechanisms in either RAP or SLOG implied some important changes in the interpreters. Thus, we chose to use a general language to simulate equational resolution and to program the control mechanisms. This language needed to be efficient enough, since we wanted to get an implementation with acceptable performances. We chose Prolog as implementation language, essentially because it includes a very efficient resolution procedure.

We use an efficient simulation method which avoid to introduce explicitly the axioms of equality. The principle of this method is well-known, and has been used in numerous approaches for introducing functions in logic programming [22, 7, 9, 11].

For this simulation, a logic program is built: to each axiom corresponds a Horn clause without equality. In [7] it is shown that, on the clauses obtained by this transformation, the standard depth-first control of Prolog provides an efficient simulation of equational narrowing with leftmost-innermost strategy. This narrowing strategy is comparable to the SLOG resolution strategy. [11] gives a method for transforming a positive conditional specification into a logic program. We use the same transformation.

As is section 2.2, some generators are distinguished among the operations of the specification. A set of generators of s sort is a set of operations $\Omega_s \subseteq \Sigma$ such that any term of s sort can be proved equal to a term made of generators only. For all s in S, we note $W_\Omega(\mathcal{X})$the term algebra generated from the operations of Ω_s and the variables of $\mathcal{X}$. The operations with results in s which do not belong to Ω_s are called "defined operations." Equational axioms, and conclusions of conditional axioms are implicitly oriented from left to right: the left-hand-side of an equation (or a conclusion of a conditional axiom) is considered as defined by the right-hand-side. Some conditions are necessary to ensure the

computational equivalence between resolution and narrowing: the top symbol of the left-hand-side of an equation (or a conclusion) must be a defined operation, and the operands of this operation must belong to $W_\Omega(\mathcal{X})$[7]. These conditions forbid, for instance, axioms between generators such as idempotence or commutativity.

We now present the transformation of a specification into a logic program. This transformation works in two steps:

1. The first step yields a set of axioms where any equation (in conclusion or precondition) only contains one occurrence of a defined operation, and it is the top symbol of its left-hand-side.
2. The second step translates these axioms into a set of Horn clauses by replacing each equality by a literal.

4.1 Transforming axioms into Horn clauses

First we give the transformation rules for the first step: they allow to transform positive conditional axioms such that in the transformed conditional axioms all the equations (both in preconditions and conclusions) are of the form:

$$f(t_1, \dots t_n) = t$$

where f is a defined operation and $t, t_1, \dots t_n$ belong to $W_\Omega(\mathcal{X})$.

Then we describe how to obtain Horn clauses from such axioms. Finally, we recall the sufficient conditions on the form of the specification which ensure the equivalence between the equational theory of the initial axioms and SLD resolution on the resulting clauses.

4.1.1 Transformation rules for the first step

Notations: ψ is a conjunction of equations, u is an occurrence in a term, and $t[u \leftarrow t']$ means: t' is the subterm of t at the occurrence u.

r1: Simplification of the right-hand-side of a conclusion

$$\frac{\psi \;\Rightarrow\; l = r[u \leftarrow f(t_1, \dots, t_n)]}{\psi \wedge f(t_1, \dots, t_n) = X \;\Rightarrow\; l = r[u \leftarrow X]}$$

where f is a defined operation and X a new variable.

r2: Elimination of a defined operation at the top of the right hand-side of an equation

$$\frac{\psi \;\wedge\; f(t_1, \dots, t_n) = g(t'_1, \dots, t'_m) \;\Rightarrow\; l = r}{\psi \;\wedge\; f(t_1, \dots, t_n) = X \;\wedge\; g(t'_1, \dots, t'_m) = X \;\Rightarrow\; l = r}$$

where f, g are some defined operations and X is a new variable.

r3: Elimination of an internal occurrence of defined operation

$$\frac{\psi \ \wedge \ t[u \leftarrow f(t_1, \ldots, t_n)] = t' \ \Rightarrow l = r}{\psi \wedge f(t_1, \ldots, t_n) = X \wedge t[u \leftarrow X] = t' \Rightarrow l = r}$$

where f is a defined operation, u is a strict occurrence in t (i.e. f is not the top symbol of t) and X is a new variable. This rule works both on the right hand-side and on the left-hand-side of an equation.

r4: Elimination of equation between terms of $W_\Omega(\mathcal{X})$

$$\frac{\psi \ \wedge \ t = t' \ \Rightarrow \ l = r}{\sigma(\psi \ \Rightarrow \ l = r)}$$

where t and t' belong to $W_\Omega(\mathcal{X})$and σ is the most general unifier of t and t' (in the free Σ-algebra).

4.1.2 Correctness of the transformation

Rules r1, r2, r3 are just specializations of the following straightforward equivalence:

$$(\forall Y)(\neg(Y = t) \vee \phi) \iff \sigma(\phi)$$

where $=$ is a congruence, Y is a variable not occurring in t, ϕ is any formula and σ is the substitution $\{Y \leftarrow t\}$.

For rule r4, the equivalence between the initial and resulting axioms is only true when the initial equational theory contains no equation between different generators. One sufficient condition ensuring the non-existence of equation between generators in the initial theory is that: there is no axiom defining a generator; and, when orienting from left to right the conclusions of the axioms, the set of axioms is a convergent (confluent and terminating) term rewrite system.

Now let us come back to the rule r4. Under this later condition, if there is no unifier of t and t', then the axiom is meaningless and can be discarded (in our system, a warning is issued to the user in this case).

We have the following properties: rules r2, r3, r4 neither remove nor create cases where rule r1 is applicable; rules r3 and r4 neither remove nor create cases where rule r2 is applicable; rule r4 neither removes nor creates cases where rule r3 is applicable. Thus we choose the following control:

1. apply r1 as long as possible,
2. apply r2 as long as possible,
3. apply r3 as long as possible,
4. apply r4 as long as possible,

It is obvious that, with this control, the transformation terminates. We give in appendix the proof that, up to the symmetry of the equalities occurring in the precondition, the transformed axioms are of the form:

$$f_1(t_{1,1}, \ldots, t_{1,n_1}) = r_1 \wedge f_m(t_{m,1}, \ldots, t_{m,n_m}) = r_m \Longrightarrow f(t_1, \ldots, t_n) = r$$

where f, f_i are defined operations and $t_{i,j}$, r_k and r belong to $W_\Omega(\mathcal{X})$.

4.1.3 Last step of the transformation

From the axioms above, a set of Horn clauses is built by replacing, in each axiom, every equation $f(t_1, \ldots, t_n) = t$ by a literal $\tilde{f}(t_1, \ldots, t_n, t)$. It means that, with each defined operation f of arity n, is associated a relation name $\tilde{f}$ of arity $n+1$. The last operand of $\tilde{f}$ corresponds to the result of f. The generators are left unchanged.

4.1.4 Examples

Let us consider the classical axioms for defining the addition operation in natural numbers, generated by zero and successor:

add(0, N) = N	$\rightarrow 1^{st}$ step $\rightarrow$	add(0, N) = N
add(s(N), M) = s(add(N, M))		add(N, M) = K $\Rightarrow$ add(s(N), M) = s(K)

where *N, M* and *K* are variables of *Nat* sort. When applying the second step of the transformation, the Horn clauses obtained for these axioms are:

$add(0, N, N).$

$add(s(N), M, s(K))$:–
  $add(N, M, K).$

The binary operation *add* becomes a ternary relation (by notation abuse we still note *add* instead of $\widetilde{add}$). We use the usual clausal notation: $\Leftarrow$ is noted ": –" and $\wedge$ is noted ",". The two clauses above are a Prolog program which defines the addition of natural numbers.

We are not only interested in the translation of axioms into Prolog. As seen in section 3, the test data selection implies the resolution of equational problems such as:

$$t_1 = u_1 \wedge \ldots \wedge t_n = u_n$$

Such a problem is transformed into a Prolog goal (negative clause) just as preconditions of conditional axioms in the transformation above.
For instance, the problem

$$add(add(0,\ Y), s(X)) = s(s(Z)) \wedge Y = s(T)$$
$$\downarrow$$
$$1^{st} \text{ step}$$
$$\downarrow$$
$$add(0,\ s(T)) = U \wedge add(U,\ s(X)) = s(s(Z)) \text{ with } \sigma : \{Y \leftarrow s(T)\}$$

where *X, Y, Z, T, U* are variables of *Nat* sort. The second step of the transformation leads to the following goal:

:– *add(0, s(T), U), add(U, s(X), s(s(Z))).*

with the substitution $\sigma : \{Y \leftarrow s(T)\}$.

4.1.5 SLD resolution versus equational reasoning

We give here some sufficient conditions on the initial specification which ensure the completeness of SLD resolution on the resulting Horn clauses with respect to the equational theory underlying the initial axioms.

These conditions were mentioned initially in [7]. They correspond to the ones given in [10] for the completeness of SLOG:

1. When orienting the axiom conclusions from left to right, the resulting conditional rewrite system is terminating (for any term, after a finite number of applications of the rewrite rules, one obtains a normal form, i.e. a term where it is impossible to apply one of the rules) and confluent (for any term, the normal form is unique).

2. There is no axiom *defining* a generator : the top symbol of the left-hand-side of an equation (or a conclusion of a conditional axiom) must be a defined operation.

3. In the left-hand-side of an equation (or a conclusion of conditional axiom), the operands are terms of $W_\Omega(\mathcal{X})$.

4. The specification is complete with respect to the generators: any ground term of W_Σ is equal to a term of W_Ω (which is unique, from the first condition).

When the specification fulfills these conditions, the SLD resolution on the resulting clauses provides a unification procedure correct and complete (for the solutions in normal form) for the equality defined by the initial axioms.

The system we have developed is based on the transformation we have presented, Prolog resolution and some specific control. Thus the specifications it can process must satisfy the four conditions above. Moreover, all the variable instances in the resulting test data sets are in normal form, i.e. belong to W_Ω. This is not restrictive since, as said in section 2, the operations in Ω generate all the values.

4.2 Using a complete search strategy

The standard control strategy in Prolog is depth-first search. It is not satisfactory for our purpose since, when there is an infinite path in the resolution tree of a problem, the solutions which are reachable by some other path may never be generated. As we want to get data sets which are valid with respect to the selection hypotheses discussed in section 3, we need a control strategy which is complete and moreover sufficiently efficient.

A well-known strategy which is complete for our purpose, is the breadth-first search. However, its implementation requires numerous copies of the problem, and complex mechanisms for aliasing of terms, variables, literal, etc.
Thus we have chosen a strategy which is a good compromise between the depth-first search and the breadth-first search: the "iterative depth-first" search consists in stating a bound k for the depth in the resolution tree. When a resolution path reaches this bound, the state of the resolution is stored and the search backtracks to try another choice of clause. Thus the resolution is complete for the solutions which are reachable by a depth less than k in the resolution. If no solution is reached for this bound, if there exist some memorized states of resolution, the process is started again from these states with a new bound $k + k'$ (one can notice that if $k = k' = 1$, it is equivalent to breadth-first search, if

$k = \infty$ it is equivalent to depth-first search). As the choice of k and k' strongly depends on the problem to be solved, we have left them as parameters of our system.

4.3 The termination problem

It is well known that Prolog programs do not always terminate, and the choice of the iterative depth-first control is far from solving this problem. For instance, even if this strategy is complete, it does not detect the unsatisfiabilities which correspond to contradictions with the axioms.

> **Example:** Given the logic program defining the addition in natural numbers, let us consider the equational problem:
>
> *add(add(s(X), Y), Z) = 0*
>
> This is an unsatisfiable problem: the sum of a natural number greater than 0 with an other natural number cannot be equal to 0. The transformation of this problem in a Prolog goal returns:
>
> :– *add(s(X), Y, T), add(T, Z, 0).*
>
> The iterative depth-first search does not terminate on this goal: the resolution of the first literal yields successively for T all the natural numbers, and each time the resolution of the second literal fails.
> However, if the initial problem is rewritten via the axioms which define *add*, it becomes:
>
> *s(add(add(X, Y), Z)) = 0*
>
> In this form, the unsatisfiability is detected at once since there is no equation between different generators in the theory.

From this example, it appears that rewriting should be used as a simplification tool before each resolution step. This simplification not only provides a way to detect some unsatisfiabilities, but it turns out to decrease in a significant way the size of the resolution tree.

Moreover, as the axioms determine a confluent and terminating rewrite system (cf section 4.1) this simplification is deterministic, and it preserves the correctness and completeness properties. Thus we have integrated this simplification in our system, after adapting it to rewriting of literals, as it is done in [11]. This mechanism was also used in the conditional narrowing algorithms of RAP and SLOG.

In the system, the command "rewrite(true)" causes a simplification by rewriting before each resolution step. If the user prefers not to use it, he can use the "rewrite(false)" command.

4.4 Random choice of clauses

As seen in section 3, a uniformity subdomain is defined by a conjunction of equations. Thus, selecting some test data under uniformity hypotheses in a given subdomain requires the resolution of a conjunction of equations. Under uniformity hypotheses, we need only one solution. But it is clear that always retaining the first solution given by the iterative depth-first control is much too deterministic for our purpose: we want an arbitrary value of the subdomain, and very often, the first solution is far from being an arbitrary value.

For instance, if an operation is recursively defined with the initial cases first, then the first returned solutions are the values corresponding to those cases. However, we do not require actual randomness in the order of the solution : any non determinism in the order of solutions is quite sufficient.

In order to ensure such a non deterministic order of the solutions, we have implemented a random choice strategy among the clauses which are applicable for a given literal.

Example: Given the operation $< : Nat \times Nat \rightarrow Boolean$ (less than), and the logic program (obtained by transformation of its axioms):

$<(X, 0, false).$

$<(0, s(X), true).$

$<(s(X), s(Y), B)$:–
  $<(X, Y, B).$

We want to solve the equation $<(X, 3) = true$. With depth-first search we get the solutions in the following order:

$X = 0;\ X = 1;\ X = 2;$

with a random-choice strategy for the choice of the clauses, we can get one of the following sequence:

$X = 1;\ X = 2;\ X = 0;$
$X = 2;\ X = 1;\ X = 0;$
$X = 0;\ X = 1;\ X = 2;$

However, all the permutations of these three solutions are not possible, since the back-tracking induces a partial order on solutions.

This is the way we have implemented selection in uniformity subdomain. In our system, this kind of control is activated and disactivated by the commands "random_choice(true)" and "random_choice(false)."

4.5 Decomposition into uniformity subdomains

It remains to provide a tool for the decomposition of the domain of an equational problem into uniformity subdomains. This tool uses some control specifications in order to stop the decomposition. As seen in Section 3, one simple way to do this decomposition is to recursively replace each defined operation occurring in the equational problem (to be decomposed) by the cases corresponding to the axioms which define it.

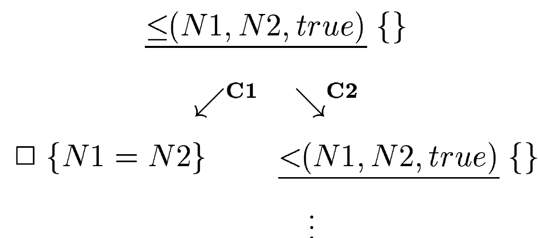
The coverage of the cases occurring in the axioms is already provided by the resolution since all the axioms are tried. Thus, what we need is a mechanism which stops the resolution of a literal when this literal defines a domain where it seems sensible to apply a uniformity hypothesis.

Example: We now consider the operation $\leq : Nat \times Nat \rightarrow Boolean$, with the corresponding logic program (cf. Section 3):

C1: $\leq(X, X, true).$
C2: $\leq(X, Y, true)$:– $<(X, Y, true).$
C3: $\leq(X, Y, false)$:– $<(Y, X, true).$

Assume we want to decompose the problem $\leq(N1, N2) = true$ until we get equations of the forms $N1 = N2$, $<(N1, N2) = true$.

The resolution of this problem builds the following search tree:



Here, each node of the tree contains a resolvent and a substitution. The edges are labeled by the clause used to solve the underlined literal in the resolvent of the origin node. □ is the empty resolvent, and {} is the identity substitution.

When continuing the construction of the tree under $<(N1, N2, true)$, one will get in the leaves of the tree, the possible values for $N1$ and $N2$.
On this very simple example, it is easy to see that if the resolution of a literal of the form $<(N1, N2, true)$ is stopped, the leaves of the tree contain the decompositions we want (i.e. $N1 = N2$ and $<(N1, N2) = true$.)

Several logic programming languages (MU-PROLOG [24], METALOG [8]) allow the programmer to specify the conditions for performing the resolution of a literal. This induces a strategy for the choice of the literal to be solved which preserves correctness and completeness. This specification is written via meta-clauses which define specific meta-predicates. The meta-clause below defines, via the *wait* meta-predicate, the conditions for delaying the resolution of its argument:

$$wait(<(A, B, true)) \;:-\; var(A), var(B).$$

This means that, if the resolvent contains a literal of the form $<(A, B, true)$ and if its operands A and B are variables, then this literal won't be selected for resolution.

We have implemented this control for the choice of the literal to be solved. Thus, the answers of the resolution procedure are couples *(substitution, constraint)*. A constraint is a list of literals waiting for subsequent resolution. Moreover, we have introduced a new mode which allows one to force the resolution of constraints. This mode is activated by the command *constraints(false)*. The command *constraints(true)* forbids the resolution of the constraints: when a resolvent contains waiting literals only, the resolution is stopped (for this path of the search tree, of course).
When this command is activated, the system builds (from the specification and the wait clauses) a set of uniformity subdomains for each axiom. These uniformity subdomains are defined by a conjunction of constraints and substitutions (where the substitutions are seen as conjunction of equations).

For efficiency reasons, we have introduced a heuristic for the choice of the literals. It gives priority to literals which do not unify with any head of clause, and to those which are unifiable to a unique head of clause.

Summing-up, the selection strategy of the literal to be solved is:

1. choose the first (in sequence) literal of the resolvent which is not unifiable with a head of clause (thus there will be a local failure),

2. then, among the literals of the resolvent which are not affected by a "wait", choose the first literal which is unifiable to a minimum of heads of clauses (deterministic literals have priority)

4.6 Strategies for regularity and uniformity hypotheses

In this subsection, we show how to use this set of tools for implementing some selection strategies compatible with the hypothesis of regularity on the generators of a sort s (Ω_s-regularity) and the uniformity hypothesis.

As in section 2, we need a canonical complexity measure on terms of s sort in W_Σ that do not contain defined operations of s sort (i.e. terms of s sort belonging to $W_{\Delta\Omega+\Sigma_1+\ldots\Sigma_n}$). We note *complexity*$_s$ this measure. Its value is the number of occurrences of a generator of s:

$$\forall f : s_1 \times \ldots \times s_n \rightarrow s \in \Omega_s,\ complexity_s(f(t_1,\ldots,t_n)) = 1 + \Sigma_{1\leq i\leq n} complexity_s(t_i)$$

It is not difficult to generate automatically the axioms defining this measure as soon as the signature is given.

Selecting a test data set corresponding to Ω_s-regularity of level n for a variable X, is reduced to the resolution of the equational problem:

$$\leq(complexity_s(X), n) = true$$

which, by transformation becomes:

$:-\ complexity_s(X, C),\ \leq(C, n, true).$

It is also possible to use the resolution procedure to select test data sets corresponding to uniformity on a sort s: given the generators of s, it is possible to construct the axioms defining a typing function: $is_a_s : s \rightarrow Boolean$

$$\forall f : s_1 \times \ldots \times s_n \rightarrow s \in \Omega_s$$

$$is_a_s_1(t_1) = true\ \wedge \ldots \wedge\ is_a_s_n(t_n) = true\ \Rightarrow\ is_a_s(f(t_1,\ldots,t_n)) = true$$

The application of a uniformity hypothesis on s for a variable X is done by computing the first solution of the goal:

$:-\ is_a_s(X,\ true).$

with a random strategy for the choice of the clauses.

Example: The generators of the *NatList* sort are *empty* $:\rightarrow$ *NatList* and *cons* : *Nat* $\times$ *NatList* $\rightarrow$ *NatList*. One builds automatically the following definition of *is_a_NatList*:

$$is_a_NatList(empty) = true$$

$$is_a_Nat(X) = true\ \wedge\ is_a_NatList(L) = true$$
$$\Rightarrow\ is_a_NatList(cons(X, L)) = true$$

Applying a uniformity hypothesis on *NatList* for a variable L consists in solving (after transformation of the axioms) the goal:

$:-\ is_a_NatList(L, true).$

The system will return, for instance:

$L = cons(5, cons(0, empty))$

In our system, the command *make_tests_tools("spec_name")* causes the automatic construction of a complexity function and of a typing function for each sort defined in the specification module *"spec_name."* The resulting axioms are stored in a specification module *spec_name_test* which uses the modules defining the required sorts and operations. For instance, the *NATLIST* specification module defines the *NatList* sort and uses the *Nat* sort. Thus, the module *NATLIST_test* uses the modules: *NATLIST, NAT_test, NAT, BOOLEAN.* These two last modules are needed for the complexity function which is of range *Nat* , and for the typing function which is of range *Boolean.*

The next (and last) section of this paper shows how to use our system for selecting test data sets for axioms of the *NATLIST* specification. We have chosen these axiom in order to exercise all the selection strategies presented here.

5 An example of test data set selection using our system

We briefly present the functionalities of our system. Then we show on an example how to use it for selecting test data sets.

5.1 Overview of the system

Our system accepts as input structured positive conditional specifications as defined in Section 2.1. The transformation of the axioms of a specification module into Horn clauses is automatically done when the user asks for compilation of this module. Note that the user never sees the internal form and he/she always works with his/her original specification. In addition, the user can control the resolution strategy defining "wait" expressions for some operations of the module *spec_name* in a module called *spec_name.ctrl* which is automatically consulted when the module *spec_name* is loaded.

The main tool of the system is an interpreter. This interpreter can work in two modes: the command mode, and the request mode.

- In the command mode, it is possible to compile specification modules, to get the "listing" of a specification or of an operation, to generate a specification module including the complexity and typing functions seen in section 4, to modify the various parameters of the resolution procedure via the commands described in the previous section.

- In the request mode, it is possible to solve equational problems of the form:

  $:-\ eq_1, \ldots, eq_n.$

  where the eq_i are equations between terms of $W_\Sigma(\mathcal{X})$ of the same sort. The solutions obtained are couples of substitution and constraint. The constraint is empty if the command *constraints(false)* has been used: there is no stop on constraints of the resolution.

The transformation of a problem into a conjunction of literals is done automatically when parsing the text of the problem.

The user can ask for one solution only with the strategy of random choice of the clauses, and without stop on constraints (i.e. the selection strategy on a uniformity subdomain) by putting a question mark as a prefix of the problem. For instance:

$:-\ ?(eq_1, \ldots, eq_n\ ).$

This construction can be used anywhere and several times in the same request:

$:-\ pb_1, ?(pb_2), pb_3.$

First, the system computes a couple $(\sigma_1, constraint_1)$ by parameterized resolution of pb_1. Second, it considers the problem $pb'_2 = \{constraint_1, \sigma_1(pb_2)\}$, and computes one solution $(\sigma_2, \emptyset)$ with the strategy of random choice of the clauses, and without stop on constraints ($constraint_1$ is solved with $\sigma_1(pb_2)$, giving no constraints as there is no stop on constraints). Finally, it returns a couple $(\sigma_3, constraint_3)$ which is a solution of

$pb3' = sigma_2(sigma_1(pb_3))$.

Before starting some test data set selection for the axioms of the *spec* module, the user ask for the creation of a module called *spec_test* which contains the definitions of the complexity and typing functions for the sorts which are defined in *spec*. He also can modify the module *spec_test.ctrl* by adding some "wait" meta-clauses.

5.2 Examples of test data set selections

We recall the defining axioms of *sorted* from the specification *NATLIST*.

$$sorted(empty) = true$$
$$sorted(cons(N1, empty)) = true$$
$$sorted(cons(N1, cons(N2, L))) = and(\ \leq(N1, N2), sorted(cons(N2, L)))$$

where $N1, N2$ belong to *Nat* and L belongs to *NatList*. The *and* and $\leq$ operations are defined as in section 3.

We now describe precisely the selection of a test data set for every defining axioms of *sorted*. As the elementary tests are equations between *Boolean* terms, there is no problem of observability: this sort is trivially observable. In non observable cases, the construction of observable contexts for the selected elementary tests can be done after this first selection phase.

We assume that the *NATLIST_test* specification has been created and loaded (*NATLIST_test.ctrl* is automatically loaded). All the required specification modules, all the complexity and typing functions are available.

5.2.1 First defining axiom of sorted

There is no variable occurring in this axiom. Thus, the only selected test is:

$$sorted(empty) = true$$

5.2.2 Second defining axiom of sorted

Only one variable occurs in this axiom. It has the imported sort *Nat*. We apply uniformity hypothesis on *Nat*.

For that, we use the typing function $is_a_Nat : Nat \rightarrow Boolean$ which was automatically generated in the *NAT_test* module (c.f. section 4.6).
We request one solution of the problem *is_a_Nat(N1)* = *true* with the strategy of random choice of the clauses:

:– $?(is_a_Nat(N1{:}Nat) = true)$.

We get as answer: $N1 = 5$.

Thus the only test for the second axiom of sorted is:

$$sorted(cons(5, empty)) = true$$

5.2.3 Third defining axiom of sorted

We now consider the last defining axiom of *sorted.*

$$sorted(cons(N1, cons(N2, L))) = and(\ \leq(N1, N2), sorted(cons(N2, L)))$$

The $N1$ and $N2$ variables are of the *Nat* sort, which is imported. The L variable is of defined sort *NatList.* Thus we apply a regularity hypothesis (of level 2) to L. For that, we use the complexity function on *NatList* in the following request:

$:- \ \leq(complexity_{NatList}(L{:}NatList), 2) = true.$

which returns two solutions:

$L = empty;$
$L = cons(X{:}Nat, empty);$

We now have two instances of the axioms, where there is no more variable of *NatList* sort:

$$sorted(cons(N1, cons(N2, empty)))$$
$$= and(\ \leq(N1, N2), sorted(cons(N2, empty)))$$

$$sorted(cons(N1, cons(N2, cons(X, empty))))$$
$$= and(\ \leq(N1, N2), sorted(cons(N2, cons(X, empty))))$$

The defining axioms of *and,* $\leq$ and *sorted* give the cases which must be covered. Thus we decompose the right hand sides of the two above instances via these axioms (as in Section 3).

The $\leq$ operation is defined in term of the $<$ operation. Let us assume that we want to cover the corresponding cases and that we want to stop the decomposition at $<$ (i.e. we do not want to analyze the cases introduced by the definition of $<$). We get three uniformity subdomains definitions (see section 3): $M = N$, $<(M, N) = true$ and $<(N, M) = true.$

The computation of this decomposition uses the "stop on constraints" mode. We need to define a "wait" meta-clause which will stop the decomposition of $<$.

```
wait(<(N:Nat, M:Nat) = B:Boolean) :-
      var(B) →
            and(or(var(N), var(M)), N =/= 0, M =/= 0)
      |     B == true →
                  var(M)
            |     var(N).
```

The meaning of $Cond1 \ \rightarrow \ Cond2 \mid Cond3$ is **if** *Cond1 is provable* **then** *prove Cond2* **else** *prove Cond3,* and $==$ and $=/=$ are respectively syntactic equality and syntactic difference.

The control of the resolution defined by this meta-clause is: if an equation of the form $<(N, M) = B$ occurs in a resolvent and if N, M, B are not instancied enough for ensuring that this equation has a finite number of solutions, then the resolution of this equation is delayed.

From the form of the defining axioms of $\leq$, it it clear that the resolution of an equation $\leq(M, N) = B$ will lead to equations $<(N, M) = true$, $<(M, N) = true$ or $M = N$ (but

$M = N$ appears only on the substitution part of the solution). The above control will block the resolution of the two remaining equations if M and N are not instancied. It is possible to give a simpler control. The interest of the one given above is that it avoids efficiently some cases of non termination; thus it makes the resolution more efficient.

Generally, the statement of the "wait" meta-clauses is not easy. These meta-clauses are not only useful to describe the selection strategy, they also allow one to define efficient control for resolution. Some work on the automatic definition of meta-clauses, in simpler cases, is reported in [25].

Given the meta-clause above, the decomposition of the first instance of the axiom is obtained, using the resolution with stop on constraints, by the request below:

$$:- \; and(\; \leq(N1{:}Nat, N2{:}Nat), sorted(cons(N2, empty))) = B{:}Boolean.$$

which returns as solutions:

$N1 = N2, B = true;$

$B = true,$
$constraint = \{<(N1, N2) = true\};$

$B = false,$
$constraint = \{<(N2, N1) = true\};$

These solutions correspond to the definitions of uniformity subdomains. To get directly arbitrary solutions in these subdomains, it is possible to use the request:

$$:- \; and(\; \leq(N1{:}Nat, N2{:}Nat), sorted(cons(N2, empty))) = B{:}Boolean, ?().$$

The ?() construct must immediately follow some equations. After the resolution of these equations, which yields some constraints and substitutions, it activates the selection strategy corresponding to uniformity hypotheses on the subdomains. Thus we get as solutions:

$N1 = 2, N2 = 2, B = true;$
$N1 = 0, N2 = s(X{:}Nat), B = true;$
$N1 = s(s(s(s(X{:}Nat)))), N2 = 3, B = false;$

The two last solutions contain a variable of *Nat* sort. As it is an imported sort, we apply a uniformity hypothesis on *Nat* via the request $?(is_a_Nat(N1) = true, is_a_Nat(N2) = true)$, which is added in the previous request:

$$:- \; and(\; \leq(N1{:}Nat, N2{:}Nat), sorted(cons(N2, empty))) = B{:}Boolean, ?(), \\ ?(is_a_Nat(N1) = true, is_a_Nat(N2) = true).$$

The ?() part of the request could be omitted. However, for a better efficiency of the resolution, it is necessary to dissociate the selection in the uniformity subdomains built by decomposition, from the uniformity hypothesis on *Nat*. The final ground solutions are:

$N1 = 5, N2 = 5, B = true;$
$N1 = 1, N2 = 3, B = true;$
$N1 = 9, N2 = 7, B = false;$

Thus we have built the following test data set for the first instance of the axiom (the instance corresponding to $complexity_{NatList}(L) = 1$)

$$
\begin{aligned}
sorted(cons(5, cons(5, empty))) &= and(\ \leq(5,5), sorted(cons(5, empty))) \\
sorted(cons(1, cons(3, empty))) &= and(\ \leq(1,3), sorted(cons(3, empty))) \\
sorted(cons(9, cons(7, empty))) &= and(\ \leq(9,7), sorted(cons(7, empty)))
\end{aligned}
$$

For the second instance, which corresponds to $complexity_{NatList}(L) = 2$, the method is similar. The request below realize a selection strategy compatible with the hypotheses used for the first instance:

$$
\begin{aligned}
:-\ & and(\ \leq(N1{:}Nat, N2{:}Nat), sorted(cons(N2, cons(X{:}Nat, empty)))) = B{:}Boolean, \\
& ?(), ?(is_a_Nat(N1) = true, is_a_Nat(N2) = true, is_a_Nat(X) = true).
\end{aligned}
$$

The resulting test data set for the second instance is:

$$
\begin{aligned}
sorted(cons(2, cons(2, cons(2, empty)))) &= and(\ \leq(2,2), sorted(cons(2, cons(2, empty)))) \\
sorted(cons(0, cons(0, cons(4, empty)))) &= and(\ \leq(0,0), sorted(cons(0, cons(4, empty)))) \\
sorted(cons(4, cons(4, cons(2, empty)))) &= and(\ \leq(4,4), sorted(cons(4, cons(2, empty)))) \\
sorted(cons(0, cons(1, cons(1, empty)))) &= and(\ \leq(0,1), sorted(cons(1, cons(1, empty)))) \\
sorted(cons(0, cons(2, cons(3, empty)))) &= and(\ \leq(0,2), sorted(cons(2, cons(3, empty)))) \\
sorted(cons(1, cons(2, cons(0, empty)))) &= and(\ \leq(1,2), sorted(cons(2, cons(0, empty)))) \\
sorted(cons(3, cons(0, cons(0, empty)))) &= and(\ \leq(3,0), sorted(cons(0, cons(0, empty)))) \\
sorted(cons(5, cons(0, cons(5, empty)))) &= and(\ \leq(5,0), sorted(cons(0, cons(5, empty)))) \\
sorted(cons(6, cons(2, cons(1, empty)))) &= and(\ \leq(6,2), sorted(cons(2, cons(1, empty))))
\end{aligned}
$$

The test data set for the third defining axiom of sorted is the union of the two data sets above.
By construction, the corresponding testing context refinement is the following: the regularity and uniformity hypotheses mentioned above are added to the hypotheses; in the test data set, the third defining axiom is replaced by the selected elementary tests above; and the oracle is the built-in equality on booleans.

When all the axioms of the *NATLIST* specification module are treated in the same way, the resulting testing context is practicable, as proved in Section 2.

One can remark that it is possible to "program" complex selection strategies with one request only. For instance, in our example, there is no necessity to separate the selection step which corresponds to regularity, as we have done for explanatory purposes. The global selection strategy, i.e. regularity of level 2, uniformity on the subdomains obtained by decomposition, uniformity on the remaining *Nat* variables can be realized by the following request:

$$
\begin{aligned}
:-\ & \leq(complexity_{NatList}(L{:}NatList), 2) = true, \\
& and(\ \leq(N1{:}Nat, N2{:}Nat), sorted(cons(N2, L))) = B{:}Boolean, \\
& ?(), ?(is_a_Nat(N1) = true, is_a_Nat(N2) = true, is_a_NatList(L) = true).
\end{aligned}
$$

The resolution of the equation $is_a_NatList(L) = true$ makes it possible to apply a selection strategy corresponding to a uniformity hypothesis on all the subterms of *Nat* sort occurring in L. The solutions of this request allow to build a test data set similar to

the previous one. Of course the instances selected by uniformity may differ.

The system we have presented here provides a sort of kernel language for programming test data selection from algebraic specifications. This kernel has been carefully designed, on sound theoretical bases: we think it is a good starting point for the development of an integrated, well interfaced environment for designing test data sets from formal specifications (as a by-product, it would be a very nice prototyping environment).

The example we have presented here is not a toy example: generating relevant test data sets for sorted lists is difficult, more difficult than most of the practical problems. At this moment, the system is experienced on a large specification, namely the embedded software of the automatic pilot system of the Lyon subway [6], and some data sets have been generated for some modules of the specification. It is too early to state conclusions, but until now, only simple requests have been necessary for this industrial example.

Comparison with previous and related works

The work reported here is the continuation of the works reported in [2, 3, 5, 17, 23] on the generation of test data sets from algebraic specifications using logic programming.

Even if there is some general agreement that using logic programming for assisting program testing is a good idea [12, 26], and that deriving test data sets from specification is interesting [27, 29, 28] , there are few other published works in this area.

A pioneering work on test methods based on algebraic specifications was [16] where the idea of exercising the axioms, via the procedures of the program under test, was first presented.

More recently, [13] used the uniformity and regularity hypotheses, as defined in [2] for testing a Unix-like directory structure against a OBJ-UMIST specification.

In [34], the idea of using constraints, first advocated in [5], is pushed further; Generic Constraint Logic Programming is experienced with an aim different from ours, which is to address incompleteness in analysis.

The main advances reported in this paper are:

- the definition of the concept of a testing context as a triple (hypotheses, test data set, oracle), and of a refinement preorder on such testing contexts;
- the introduction of oracle hypotheses, and more generally of a satisfactory way of coping with the oracle problem (it was not the case in [3]);
- an attempt to enlarge the class of formal specifications considered. As shown in section 1, this generalization works well but for the unbias properties and the oracle problem which are dealt with in the specialized framework of algebraic specifications. We intend to continue in this direction;
- the replacement of the notion of generation by the notion of selection: it allows a very nice justification of the way the test data sets are refined;
- the existence of a specialized system which allows to program various selection strategies, which are justified theoretically.

Acknowledgements
This work has been partially supported by the Meteor Esprit Project and the PRC "Programmation et Outils pour l'Intelligence Artificielle." It is a pleasure to thank Laurent Fribourg for numerous helpful discussions.

Appendix: Form of the axioms resulting from the first step of the transformation

We have to prove that the axioms resulting from the transformation presented in section 4.1 are of the form:

$$f_1(t_{1,1},\ldots,t_{1,n_1}) = r_1 \wedge f_m(t_{m,1},\ldots,t_{m,n_m}) = r_m \Longrightarrow f(t_1,\ldots,t_n) = r$$

where f, f_i are defined operations and $t_{i,j}$, r_k and r belong to $W_\Omega(\mathcal{X})$.

- The form of the left hand-side of the conclusion is a direct consequence of the form of the initial axiom.
- For the right hand-side of the conclusion, r, if it contains an occurrence of a defined operation, r1 is applicable; this is impossible since the control apply r1 while possible.
- For the equalities of the precondition: let us note $s_1 = s_2$ such an equality.

  1. If both s_1 and s_2 are $W_\Omega(\mathcal{X})$ terms, r4 is applicable.
  2. Else, if either s_1 or s_2 contains a strict occurrence of a defined operation, r3 is applicable.
  3. Else, if both the top symbols of s_1 and s_2 are defined operations, r2 is applicable.

  In these three cases there is a contradiction with the control. Thus either s_1 or s_2 belong to $W_\Omega(\mathcal{X})$. Then, up to the symmetry of these equalities, all the equalities in the precondition are of the defined form.

References

[1] Goguen J. Thatcher J. Wagner E. *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.

[2] Bougé L. Choquet N. Fribourg L. Gaudel M. C. *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. Also: Internal report LRI n^o176.

[3] Bougé L. Choquet N. Fribourg L. Gaudel M. C. *Test sets generation from algebraic specifications using logic programming* Journal of Systems and Software Vol 6, n^o4, pp.343-360, November 1986. Also: Internal report LRI n^o240.

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

[5] Choquet N. *Test data generation using a PROLOG with constraints* Workshop on Software Testing, Banff Canada, IEEE Catalog Number 86TH0144-6, pp 132-141, July 1986.

[6] Dauchy P., Marre B. *Test data selection from algebraic specifications: application to an automatic subway module.* ESEC'91, 3rd European Software Engineering Conference. Proceedings LNCS 550, Springer-Verlag, A.van Lamsweerde, A.Fugetta Eds, pp. 80-100. Also LRI report n° 638, LRI, Université Paris-sud, Orsay, France, january 1991.

[7] Deransart P. *An Operational Algebraic Semantics of PROLOG Programs* Proc. Programmation en Logique, Perros-Guirrec, CNET-Lannion, March 83.

[8] Dincbas M. Le Pape J.P. *Metacontrol in logic programs in METALOG* 5th Generation Conf., Tokyo, November 1984.

[9] Van Emden M. H., Yukawa K. *Logic Programming with Equations* J. Logic Programming (4), pp. 265-268, 1987.

[10] Fribourg L. *SLOG, a Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting* International Symposium on Logic Programming, Boston, July 1985.

[11] Fribourg L. *Prolog with Simplification* Programming of Future Generation Computers. Fuchi, Nivat Ed. Elsevier Science Publishers B.V. (North Holland), 1988.

[12] Gerhart S. *Software Engineering Perspectives on Prolog* Technical report TR-85-13, Wang Institute of Graduate Studies, July 1985.

[13] Gerrard C. P., Coleman D., Gallimore R. *Formal Specification and Design Time Testing* Software Science ltd, technical report, June 1985, IEEE trans. on Software Engineering, vol. 16, n°1, january 1990.

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

[15] Geser, Hussmann *Experiences with the RAP system – a specification interpreter combining term rewriting and resolution techniques* Proc. ESOP 86 Conf., LNCS 213, pp 339-350, 1986.

[16] Gannon J. McMullin P. Hamlet R. *Data-Abstraction Implementation, Specification, and Testing* ACM transactions on Programming Languages and Systems, Vol 3, n° 3, pp.211-223, July 1981.

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

[18] Hennicker R. *Observational implementation of algebraic specifications* Acta Informatica, vol.28, n°3, pp.187-230, 1991.

[19] Hussmann *Unification in Conditional-Equational Theories* Technical Report MIP-8502, U. passau, January 1985, and short version in Proc. EUROCAL 85 Conf., Linz.

[20] Kamin S. *Final Data Types and Their Specification* ACM Transactions on Programming Languages and Systems (5), pp.97-123, 1983.

[21] Kaplan S. *Conditional rewrite rules* Theoretical Computer Science (33), pp. 175-193, December 1984.

[22] Kowalski R. *Logic Programming* Proc. IFIP 1983, pp. 133-145.

[23] Marre B. *Génération automatique de jeux de tests, une solution : Spécifications algébriques et Programmation logique* Proc.Programmation en Logique, Tregastel, CNET-Lannion, pp.213-236, May 1989.

[24] Naish L. *An introduction to MU-PROLOG* Technical report, 82/2, Dept. of Computer Science, U. of Melbourne, 1982.

[25] Naish L. *Negation and Control in Prolog* LNCS 238, (Springer-Verlag), 1985.

[26] Pesch H. Schnupp P. Schaller H. Spirk A.P. *Test Case Generation using Prolog* ICSE 8, London, 1985, pp 252-258.

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

[28] Rubaux J.P. *Rapport final d'étude du poste de generation de test ASA* RATP Paris, Direction des Equipements Electriques, TT-SEL/89-183/JPR.

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

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

[31] Sannella D., Tarlecki A. *On observational equivalence and algebraic specification* Journal of Computer and System Sciences 34, pp.150-178, 1987.

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

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

[34] Wild C *The use of Generic Constraint Logic Programming for Software Testing and Analysis* Comp. Sc. Technical Report Series, no 88-02, Dept of Computer Science, Old Dominion University, Norfolk, VA, February 1988.
Extended abstract in Proc. 2nd ACM-IEEE Workshop on Software Testing, Verification, and Analysis, Banff, July 1988.