APPLICATION OF PROLOG TO TEST SETS GENERATION FROM ALGEBRAIC SPECIFICATIONS

L. Bougé (*), N. Choquet (**), L. Fribourg (**), M.C. Gaudel (***)

(*) LITP, Université Paris 7, 2 place Jussieu, 75251 Paris Cedex 05, France
(**) Laboratoires de Marcoussis-C.G.E, route de Nozay, 91460 Marcoussis, France
(***) LRI, Université Paris Sud, 91405 Orsay, France

ABSTRACT

We present a method and a tool for generating test sets from algebraic data type specifications. We give formal definitions of the basic concepts required in our approach of functional testing. Then we discuss the problem of testing algebraic data types implementations. This allows the introduction of additional hypotheses and thus the description of an effective method for generating test sets. The method can be improved by using PROLOG. Indeed, it turns out that PROLOG is a very well suited tool for generating test sets in this context. Applicability of the method is discussed and a complete example is given.

INTRODUCTION

Functional or "black-box" testing has been recognized for a long time as an important aspect of software validation [Howd 80], [ABC 82]. It is especially important for large-sized, long-lived systems for which successive versions have to be delivered. In this case, non-regression testing may be long, difficult and expensive. It should depend only on the functional specifications of the system [Paul 83].

However, most of the studies on test data generation have focused on program dependent testing [Haml 75], [Clar 76], since it was possible to use the properties of a formal object: the program. Of course such an approach is necessary but not sufficient [Gour 83]. The emergence of formal specification methods makes it possible to found functional testing on a rigorous basis. In this paper we present a method and a tool for generating test sets from algebraic data type specifications. We consider hierarchical, positive conditional specifications with preconditions. More precisely, we study how to test an implementation of a data type against a property (an axiom) which is required by the specification. The formal specification is used as a guideline to produce relevant test data.

As asserted in [BA 82], it is especially dangerous, when studying testing and correctness to use informal definitions, even if they seem obvious. For instance, it is shown in [BA 82] that two different, but rather similar, formal definitions of what an "adequate test set" is, lead to very dissimilar issues.

The first part of this paper is therefore devoted to formal definitions of several concepts: first we give the fundamental properties of what we call a **collection of test sets**; then we state the hypotheses which are assumed during the testing process and ensure the **acceptability** of the considered collections of test sets. This notion of acceptability is

defined and discussed with respect to the classical properties required for test selection criteria [GG 75]: reliability, validity and lack of bias.
In the second part, we show that using algebraic data types allows the introduction of further hypotheses and enables the test sets generation.
The third part describes how to improve this method using PROLOG. It turns out that PROLOG is a very well suited tool for generating test sets in this context. In particular it automatically provides partitions of the domains of the variables. Of course, the use of PROLOG somewhat limits the kinds of specifications and properties which can be considered. However these limitations could be alleviated by extensions of PROLOG (e.g. [Komo 80], [Nais 82], [DJ 84], [GM 84], [Frib 84], etc.)

1. BASIC NOTIONS IN TESTING THEORY

It is now widely recognized that a sound theory of testing must focus not only on the question "What is a test?". Goodenough and Gerhart [GG 75] took a decisive step forward by enlarging this question to "What is a test criterion?". A criterion makes it possible to decide whether a given set of data can be validly considered as a test. [Boug 82] suggests one more step: one globally considers the whole testing process, or at least a model of it. Indeed, the properties of test data are not independent from the method used to perform testing and to handle results.

1.1. DEFINITIONS

The **Testing Process Diagram** (fig. 1) is an attempt to model the sequence of operations that take place between the problem definition ("Does this system validate this property?") and the conclusion ("Yes, as far as I can assure it" or "No, definitely not").

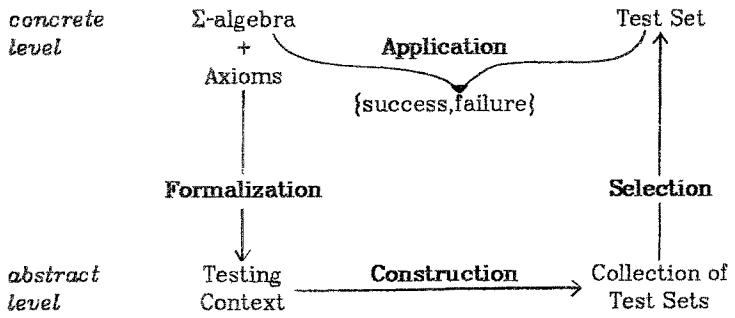

fig. 1 Testing Process Diagram

Here we consider an implementation to be a Σ-algebra (i.e. a set of operations on some sets of values) and we want to test whether this algebra satisfies a set of axioms. However, the definitions we give are general.
The problem to be solved is formally stated as a **Testing Context** *C*. It is mainly made up of the property *A* to be tested (the axioms) and previous knowledge about the system under test (the given algebra). The idea of testing a partially defined/known object has recently been introduced by De Millo et al. [BDLS 80], [Budd 81], [Howd 82]. We think it is quite fundamental. One never knows the object under test perfectly. One only knows that it satisfies some properties *H*, which come from the context or from

previous tests or proofs.
In the case of a program, one knows the semantics of its elementary components, but not its behavior over its whole domain. In the case of a Σ-algebra, one knows the behavior of lower-level type operations (Integer, Boolean...), perhaps some axioms which are a priori satisfied by this algebra (using some previous validation results), but certainly not the whole functional behavior of the algebra. Hopefully the algebra presently under test satisfies properties *H*.

Once a testing context *C* has been stated, the next step in the testing process is to build a **collection of test sets** from *C*. Such a collection is a family of test sets (T_n) indexed by positive integers. We require this family to be **asymptotically reliable**: it means that if T_{n+1} is successful then T_n is successful. This new notion is introduced to express the idea that increasing the size of a test set increases the quality of testing. In fact, here the classical concept of **test selection criterion** is replaced by the notion of collection: the (T_n) are all the test sets possibly selected by the given criterion. We will later show that the qualities of a criterion can be expressed as constraints on the sequence (T_n).

In this paper, testing is trying to answer a finite number of elementary questions of the form (at least in the case of an equational specification)

"Does the Σ-algebra X satisfy equation t=t' ?"

The straightforward way to do this is to consider all the possible instantiations of the variables occurring in the equation and to compute both sides of the equation in *X*. Each T_n is a subset of the possible instantiations of the equation. We shall note $T_n = \{t_i=t'_i\}$.

The last step to perform in the testing process is to select a test set T_n and to compute it in the Σ-algebra. If all the equations $t_i=t'_i$ are satisfied then the result of the testing process is "success"; if not, it is "failure" and one can conclude that the Σ-algebra, i.e. the data type implementation, is faulty.
The criterion for choosing n is of course strongly related to cost/quality requirements.

1.2. FUNDAMENTAL PROPERTIES OF A COLLECTION OF TEST SETS

The significance of the testing process conclusion is highly related to the quality of the test sets collection. Goodenough and Gerhart required **reliability** and **validity** for test selection criteria, leading to **ideality**. They proved that the success of the application of a test set selected by an ideal criterion implies correctness. Here we follow a similar approach.

Reliability is a consistency requirement. A collection of test sets is said to be reliable if a test set of higher index is "better" than a test set of lower index whatever potential Σ-algebra is considered. This can be formally written as follows

$$\forall n \in \mathbf{N},\ (H \cup T_{n+1}) \vdash T_n$$

This requirement is slightly weaker than Goodenough and Gerhart's and, is called **asymptotic reliability**. It captures the fact that testing is fundamentally an incremental process.

Validity means that any incorrect behavior will be revealed by some test data in some T_n, i.e.

$$(H \cup (\cup_n T_n)) \vdash A$$

If testing is successful using all test sets T_n then the algebra fulfills the required properties. A collection which satisfies this property is said to be **asymptotically valid**.

One more property is required for test sets collections. It is the **lack of bias**. Any correct algebra should pass any test set T_n. It is precisely the converse of validity:

$$\forall n \in \mathbf{N},\ (H \cup A) \vdash T_n$$

If a test using the test set T_p selected from the collection (T_n) fails then the algebra does not satisfy the axioms. It turns out that most "natural" test selection criteria satisfy this property (for instance, see Goodenough and Gerhart's criteria). A similar property is considered in [Gour 83].

A (asymptotically) reliable, (asymptotically) valid and unbiased collection of test sets is said to be **acceptable**. These three properties ensure that the higher the index of the selected test set, the better the conclusion of the testing process. The existence of an acceptable collection of test sets is strongly related to the properties of A and H (see [Boug 82] and [Boug 83] for details). An interesting feature of algebraic data type specifications is that they do not involve any existential quantifiers. This ensures the existence of such a collection under standard extra hypotheses to be listed below.

1.3. REGULARITY AND UNIFORMITY HYPOTHESES

The problem of testing axioms for a Σ-algebra is thus reduced to seeking an acceptable collection of test sets for a testing context. It is only possible if some powerful assumptions on the Σ-algebra are available. Such assumptions are left implicit in most testing methods. For notational convenience, let us assume hereafter that we are testing an axiom of the form t(x)=t'(x). Thus, in the following, test sets T_n are sets of instantiations of this equation.

1.3.1. Regularity hypotheses

Let us assume it is possible, in some way, to associate a level of complexity with each element of Σ-algebra carriers. The **regularity hypothesis** states that the axiom under test behaves regularly with respect to this measure. If it holds for any object of complexity less than k (k being a parameter), then it holds for any object.

$$\forall x\ (\text{complexity}(x) \leq k \Rightarrow t(x)=t'(x)) \Rightarrow \forall x\ (t(x)=t'(x))$$

Typically, complexity will be the length of a representative Σ-term denoting an object. In the case of program testing, it corresponds to the computation complexity. Thus regularity hypotheses reflect path analysis testing strategies [Howd 76], [WHH 80].

1.3.2. Uniformity hypotheses

If no complexity measure is available , we are faced with the well-known problem of partitioning variable domains in such a way that the axiom under test "behaves uniformly" on these subdomains. Formally speaking, it means that the following **uniformity hypothesis** is satisfied for each subdomain

$$\exists x\ (t(x)=t'(x)) \Rightarrow \forall x\ (t(x)=t'(x))$$

It is modelled by introducing a **new** constant c of suitable type, a **meta-constant**.

The value of such a constant is intuitively a random value of the subdomain. The hypothesis can thus be expressed as well by

$$(t(c)=t'(c)) \Rightarrow \forall x\ (t(x)=t'(x))$$

This typically leads to random testing strategies and subdomain testing strategies [WC 80], [ZW 81].

2. APPLICATION OF THE THEORY TO ALGEBRAIC DATA TYPE SPECIFICATION TESTING

We now focus on the specific kind of testing we are dealing with: testing a data type implementation against an algebraic data type specification.

2.1. THE PROBLEM

Algebraic specifications of data types are widely recognized as a useful formal specification method. See for instance [BH 85]. A specification is given by
a many-sorted signature Σ, i.e. a list of functional symbols on a set of sorts S, and
a set of Σ-axioms E.
The problem is: are the axioms of E satisfied by a given Σ-algebra X.

---

specif queue-of-int =
enrich bool, int *by*
sort queue;
operations

emptyq :		->	queue
append :	queue * int	->	queue
remove :	queue	->	queue
first :	queue	->	int
isempty :	queue	->	bool

variables
Q,Q': queue
I: int
precondition
pre(first,Q) = (isempty(Q)=false)
axioms
A1: isempty(emptyq)=true
A2: isempty(append(Q,I))=false
A3: remove(emptyq)=emptyq
A4: isempty(Q)=true => remove(append(Q,I))=emptyq
A5: isempty(Q)=false => remove(append(Q,I))=append(remove(Q),I)
A6: isempty(Q)=true => first(append(Q,I))=I
A7: isempty(Q) = false => first(append(Q,I)) = first(Q)

fig.2 Specification of Queue of Integers

Usually, one deals with hierarchical abstract data types [GH 78], [Bido 81], [BDPP 83]. A sort of interest s_i is distinguished in S, and Σ is accordingly split into signature Σ_i (i standing for interest) and Σ_p (p standing for primitive). Σ_i contains operations where at least one input or output variable is of sort s_i.
Hierarchical algebraic data types induce in a natural way a similar structure into the

testing process: lower level modules are first tested against their specification, then higher level ones. Of course, the testing of higher level modules can use the fact that lower level modules were successfully tested.

We consider only a restricted class of algebraic specifications characterized as follows:
hierarchical specifications;
a predefined boolean specification with two constants, **true** and **false**;
preconditions on operators and conditional equations, with a restricted form.
Premises of preconditions and conditional equations are restricted to be **boolean equations**, i.e. equations of the form **t=true** or **t=false** where t is a term of boolean sort. The reasons for this restriction appear in part 3. An example of such a specification is given on figure 2.
Conditional axioms such as A6 or A7 of figure 2 are valid for an algebra *X* if for any instantiation of Q and I which satisfies the preconditions and premises, both sides of the conclusion equation yield the same value in *X*.

2.2. BASIC HYPOTHESES FOR TEST SETS GENERATION

The basic assumption for test construction in such a framework is the **Correlation principle**

"There exists a narrow correlation between specification structure and implementation structure."

This is a postulate. It may definitely not be the case for our specific algebra *X*. In fact, because of the increasing use of construction methods guided by specifications [B2G3 84], using top-down, bottom-up, stepwise refinement, this principle is more and more valid as time goes. This principle is closed to the so-called competent programmer hypothesis [Budd 81]. It is more or less assumed by most of testing methodologies.

This principle is used to derive the following three hypotheses.

Finitely generated and non-trivial algebras

The first hypothesis restricts the considered algebras to be finitely generated with respect to hierarchy [WPP 83], [SW 83]. It means that any element of *X* can be denoted abstractly by application of operations of Σ_i (the operations of interest) to elements of lower sort. In the queue example of figure 2, any queue element of *X* can be then obtained as a sequence of remove and append operations on emptyq. This hypothesis states that the specification under test covers all parts of *X*. Any element of *X* can thus be denoted as a formal term of the specification.

It is necessary to avoid trivial algebras, i.e. algebras where any property is satisfied. We assume therefore that the implementation of predefined booleans satisfies the property **true≠false**.

Uniformity hypotheses for lower sorts

The specification under test is hierarchical. At testing time, lower level modules already exist (or can be simulated) and have been successfully tested against their specification. If the specification is hierarchically consistent then the correctness of lower types is preserved. One is therefore entitled to set uniformity hypotheses about lower level domains. For instance in the queue specification (see figure 2) the values of integer operands are not significant.

Regularity hypothesis for the sort of interest

The sort of interest is the actual subject of the testing process. Because algebras are finitely generated, the computational complexity of objects is directly connected to the syntactical complexity of their denotation. A possible complexity measure of an element x of X is then the length of the smallest Σ_i-term denoting x. Having in mind such a complexity measure, a regularity hypothesis directly arises. If the implementation works in all simpler cases, it will do so in more complex cases. The distinction between "simpler" and "more complex" cases is stated by choosing a complexity level k. We call k the level of the test set.

2.3. TEST SETS GENERATION ALGORITHM FOR EQUATIONAL SPECIFICATIONS

Consider an equational axiom of the form

$$t(x_1, ..., x_m)=t'(x_1, ..., x_m)$$

both sides being terms of the sort of interest. Under the three hypotheses above, we can describe an acceptable collection of test sets (T_n). Test set T_k is the **finite** set $\{t_i=t'_i\}$ of all the closed instantiations of the axiom under test obtained as follows.

Instanciation algorithm (equational case)

for i = 1 **to** m **do**
  if x_i is a variable of the sort of interest
  then instantiate it by **all** the terms of size less than k
    which contain no variable of the sort of interest
done

for each of the resulting instanciated equations **do**
  for each variable y **do**
    instantiate y by a **new** meta-constant c, one for each uniformity
    subdomain of the sort of y
  done
done

Running test set T_k simply consists of checking the validity of all its totally instantiated equations $t_i=t'_i$ on the Σ-algebra under test X. Because no variables are left, this is simply done by computing each side of the equation and checking that both yield the same value. When computing, random values of the corresponding subdomain are substituted for meta-constants.

Consider the case where a set of **constructors** (see section 3.2) is given together with the specification of the type of interest. Hypotheses can then be strengthened by assuming that X is actually finitely generated with respect to those constructors. Instantiation may thus be limited to those terms of size less than k which are combinations of constructors. The number of generated instantiations is then considerably decreased. This corresponds precisely to optimizing a test set by discarding redundant tests. This optimization is usually left implicit in testing methodologies.

Our specifications generally contain conditional axioms (see fig. 2). It may then happen that no term of size less than level k validates the premise of some axiom. It would thus be declared valid because it is vacuously satisfied for all those terms. Some check must therefore be added to ensure that all axioms have actually be tested (premises are satisfied in enough representative cases). However, another

more efficient approach is to selectively generate terms that validate some premise. This is the subject of the next section.

3. TEST SETS GENERATION FOR CONDITIONAL AXIOMS

3.1. Use of PROLOG to satisfy relations

A PROLOG program is made up of Horn clauses. A Horn clause is a conditional formula made of a head part and a body part; the head part is a relation P over terms, and the body part is a list of conditions under which the head part is true.
A PROLOG interpreter uses automatic deduction methods (resolution) to compute the terms which satisfy a relation characterized by the clauses of the program.
When a relation P(X) is written in PROLOG, then given the goal: ?-P(X), the interpreter instantiates X with the terms satisfying P.

Example:
What are the values of X such that X>2?
Booleans are defined by true and false.
Integers are built on 0 and succ, with in addition an operator le: int * int --> bool, defined in PROLOG by:

```
le(0,X,true).
le(succ(X),0,false).
le(succ(X),succ(Y),B):- le(X,Y,B).
```

Given the goal

```
?- le(X,succ(succ(0)),false).
```

the interpreter provides the general solution:

```
X = succ(succ(succ(Y))),
```

where Y takes any value.

Theoretically, the resolution strategy underlying PROLOG provides all the solutions for a goal [Clar 77].
A solution computed by the PROLOG interpreter is either a fully instantiated term or a term containing variables; in the latter case, the computed term embodies a whole class of solutions, since any instantiation of the computed term is a particular solution of the goal. This PROLOG computation feature is used hereafter.
One advantage of PROLOG in our framework is the handling of conditional axioms. However some limitations, due to the fact that equality is not handled, still exist. But some propositions are presently being submitted to alleviate this restriction [DJ 84],[GM 84],[Frib 84].

3.2. Converting a specification into PROLOG

A specification which satisfies the syntactical restrictions we have introduced on algebraic specifications in part 2.1 can generally be translated into PROLOG, (a similar translation is developed in [HS 85]).
Axioms are viewed as definitions of function symbols. Syntactically, a function symbol f is defined by a set of axioms of the form :

$$\mathbf{(a(u)=true)\ \&\ (b(u')=false) \Rightarrow f(v)=g(w)}, \qquad (*)$$

where f and g are function symbols; u, u', v, w are vectors of terms, and a, b are symbols of boolean functions; a(u)=true and b(u')=false are the **constraint equations**.

Axioms are thus implicitly oriented. The symbol f appearing in the axiom above is said to be **specified**. A function symbol specified by no axiom of the specification is called a

basic symbol or **constructor**. In this paper, we assume that there is no equation between constructors.

Axioms are translated into **Horn clauses**. The first step is to modify the signature. All the function symbols of arity n specified by axioms in the original specification are replaced in the new specification by relation symbols of arity n+1. For instance, the **operator** remove(q) (see fig. 2) becomes the **relation** remove(Q1,Q2). The only remaining terms in the translation are those formed with constructors and variables only. For instance, the constructors of the queue type are emptyq and append. Terms like emptyq and append(Q,I) are preserved.

In a second step, **axioms** are turned into **Horn clauses**. For simplicity, consider an axiom such as (*), where u, u', v are made of constructors and variables only. It becomes:

f(v,Z) :- a(u,true), b(u',false), R(X,Z),

where X is the set of variables appearing in v, and R(X,Z) expresses in a relation form the functional equation: Z=g(w). Intuitively speaking, Z is no more than an intermediate result.
When u, u', or v contain derived operators, there is a preliminary transformation in order to reduce this case to the previous one.

The last step is to plug possible **preconditions** on f into the Horn clause. If there is pre(f,x) = p(x) in the original specification, then the final clause is

f(v,z) :- a(u,true), b(u',false), p(v,true), R(X,Z).

An example is given in figure 3.

```
C1: isempty(emptyq,true).
C2: isempty(append(Q,I),false).
C3: remove(emptyq,emptyq).
C4: remove(append(Q,I),emptyq):-isempty(Q,true).
C5: remove(append(Q,I),append(Q',I)):- isempty(Q,false),remove(Q,Q').
C6: first(append(Q,I),I):- isempty(append(Q,I),false),isempty(Q,true).
C7: first(append(Q,I),J):- isempty(append(Q,I),false),isempty(Q,false),first(Q,J).
```

fig.3 Translation of the queue specification into PROLOG

3.3. Constraint-driven generation of terms

Generally speaking, each clause C derived from the specification is of the form:
f(I,O):- A(I), R(I,O), where A(I) expresses the preconditions and the premises of the original axiom. Terms satisfying A(I) are precisely those needed at the end of section 2.3 to test this original axiom in a non-trivial way. These terms are obtained by submitting the goal **?- A(I)** to PROLOG.
Consider for instance clause C5 in fig.3 . In the queue example, the relevant terms are obtained by submitting ?- isempty(Q,false). This yields the general answer
Q = append(Q',I).

This example is a simple one. Let us consider a more interesting example - *insertion into a sorted list* -, which is completely given in appendix.
Consider the clause C6 in this new example.

C6: insert(ap(L,X),Y,ap(ap(L,X),Y)):- sorted(ap(L,X),true),le(X,Y,true).

The constraints here are : sorted(ap(L,X),true) and le(X,Y,true),
where le (≤) is described by:

```
le(0,X,true).
le(succ(X),0,false).
le(succ(X),succ(Y),B):-le(X,Y,B).
```

and sorted is described by:

```
sorted(el,true).
sorted(ap(el,X),true).
sorted(ap(ap(L,X),Y),B):- le(X,Y,true),sorted(ap(L,X),B).
sorted(ap(ap(L,X),Y),false):- le(X,Y,false).
```

The constraint on C6 is solved with the goal

?-sorted(ap(L,X),true),le(X,Y,true). If we limit L to lists of length 1, L=ap(el,I), the goal becomes

?- sorted(ap(ap(el,I),X),true),le(X,Y,true). We obtain the answers:

I = 0, X = 0, Y = _;
I = 0, X = succ(0), Y = succ(_);
I = succ(0), X = succ(0), Y = succ(_);
etc ...

These answers correspond to the triples <I,X,Y> of terms of the form:

$< succ^m(0) , succ^n(0), succ^n(\_) >$, with $0 \leq m \leq n$

A standard PROLOG interpreter, using a depth-first strategy, will go into an infinite branch. It will generate a collection of solutions with increasing complexity, satisfying the goal. Unfortunately some branches might be ignored. If we stop execution after a finite number of steps, we do not have all the terms t such that complexity(t)≤k.
To get an acceptable test sets collection, all branches must be explored. This requires active control of the search strategy . This control is provided in PROLOG extensions such as MU-PROLOG and METALOG [Nais 82] [DL 84]. It is then possible to get all the terms of length less than some bound k.
PROLOG may provide terms with variables. These terms correspond to a class of solutions. Thus PROLOG automatically provides some uniformity hypotheses. Variables correspond to meta-constants (see section 2.3) .

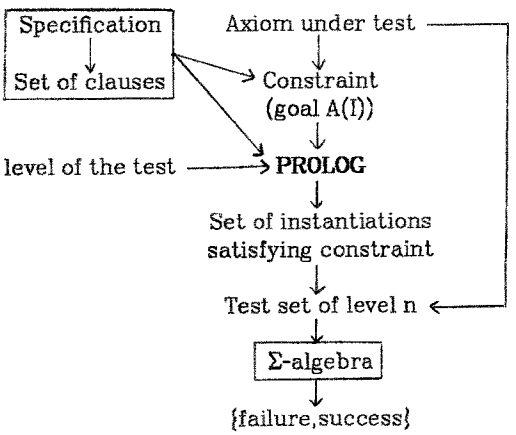

fig.4 Diagram of test sets generation

The method is summarized in figure 4. For each clause we generate terms satisfying the constraints of the clause: PROLOG will generate all of them, provided we can control the exploration of infinite branches. We take this set of terms as a domain on which we make regularity hypotheses. PROLOG helps us to partition it into uniformity sub-domains, from which we extract test data through the use of meta-constants. Thus, the definition domain has been partitionned into regularity and uniformity sub-domains. The generated test sets collection is "acceptable" according to the theory described in section 1.3, provided the search strategy is complete.

CONCLUSION

The idea of using PROLOG, or some extension of PROLOG, to generate test sets seems promising. In this paper we suggest a method which is based on the theory of testing presented in section 1. This method is applicable provided the hypotheses of section 1 are satisfied; the specifications can be translated into PROLOG; and it is possible to control the search strategy in the PROLOG interpreter. Algebraic specifications are especially well suited to such an application since it is possible to define some restrictions on them, such as those presented in section 2, so that the two first requirements of the method are satisfied.
This paper applies the method to positive conditional algebraic specifications using search strategy control provided by METALOG. The method was applied to test real-time software such as alternating bit protocol implementations and telephone switching modules. PROLOG provides a partition into uniformity domains. METALOG is very convenient for defining general search strategies which correspond to regularity hypotheses: when working with a new specification it is only necessary to define the complexity of the test data for the sort of interest.
However, to be generally applicable, this method must be improved in two directions. First the cost in time and space of PROLOG implementations must be decreased. The main limitations experienced using the examples were those of the computation time and memory overflows.
Second the class of considered specifications must be enlarged as far as possible in order to avoid rewriting the specifications for generating test sets. There is an inherent limitation to the method since the tested properties must ensure the existence of an acceptable test sets collection. Such is not the case if there is an existential quantifier in the property. However it would be possible to consider full positive conditional axioms if equality were handled by PROLOG. We are working on such a PROLOG with equality, which extends the class of specifications under consideration and allows equations between constructors.

Acknowledgements

We are very much indebted to Michel Bidoit who suggested the use of PROLOG and to Jan Komorowski for his support. M.C. Gaudel thanks the members of IFIP-WG2.2 for fruitful discussions and comments.
This research is supported by CIT-Alcatel.

REFERENCES

[ABC 82] W.R. Adrion,M.A. Branstad and J.C. Cherniavsky, "Validation, verification and testing of computer software", ACM Comp. Surv. 14, 2 (June 82).

[Bido 81] M. Bidoit, "Putting together fair presentations of abstract data types into structured specifications", Rept. No. 15/81, GRECO, France (1981).

[Boug 82] L. Bougé, "Modélisation de la notion de test de programme; application à la production de jeux de tests", Thèse de 3ème cycle, Université Paris 6, Paris (Oct. 1982).

[Boug 83] L. Bougé, "A proposition for a theory of testing: an abstract approach to the testing process", Rept. No. PB-160, DAIMI, Aarhus University, Denmark (may 83), to appear in Theor. Comp. Science.

[Budd 81] T.A. Budd "Mutation analysis: ideas, examples, problems and prospects", in: Computer Program Testing, B. Chandrasekran and S. Radicchi, eds. (North-Holland, 1981) 129-148.

[BA 82] T.A. Budd and D. Angluin, "Two notions of correctness and their relation to testing", Acta Informatica 18 (1982) 31-45.

[BDLS 80] T.A. Budd, R.A. De Millo, R.J. Lipton and F.G. Sayward, "Theoretical and empirical studies on using program mutation to test the functional correctness of programs", Proc. 7th Ann. ACM Symp. Princ. Prog. Lang., Las Vegas (Jan. 1980) 220-233.

[BH 85] B. Biebow and J. Hagelstein, "Algebraic specification of synchronization and errors: a telephonic example", Proc. Coll. Soft. Eng., Berlin (1985), this volume.

[B2G3 84] M. Bidoit, B. Biebow, M.C. Gaudel, D. Gresse and G. Guiho, "Exception handling: formal specification and systematic program construction", Proc. Int. Conf. Soft. Eng., Orlando, Florida (1984).

[Clar 77] K.L. Clark, "Predicate logic as a computational formalism", Research Rept., Dept. of Computing, Imperial College, London (1977).

[Clar 76] L. Clarke, "A system to generate test data and symbollicaly execute programs", IEEE Trans. Soft. Eng. SE-2, 3 (1976) 215-222.

[DJ 84] N. Dershowitz and N.A. Josephson, "Logic programming by completion", Proc. 2nd Int. Logic Programming Conf., Uppsala, Sweden (July 1984) 313-320.

[DL 84] M. Dincbas and J.P. Le Pape, "Metacontrol in logic programs in METALOG", 5th Generation Conf., Tokyo, Japan (Nov. 1984).

[Frib 84] L. Fribourg, "Oriented equational clauses as a programming language", J. Logic Programming, 2 (Oct. 1984) 165-177.

[Gour 83] J.S. Gourlay, "A mathematical framework for the investigation of testing", IEEE trans. Soft. Eng. SE-9, 6 (1983).

[GG 75] J.B. Goodenough and S.L. Gerhart, "Toward a theory of test data selection", IEEE Trans. Soft. Eng. SE-1, 2 (1975).

[GH 78] J. Guttag and J. Horning, "The algebraic specification of abstract data types", Acta Informatica 10, 1 (1978).

[GM 84] J. Goguen and J. Meseguer, "Equality, types, modules and generics for logic programming", Proc. 2nd Int. Logic Programming Conf., Uppsala, Sweden (July 1984) 115-125.

[Haml 75] R.G. Hamlet, "Testing programs with finite sets of data", Rept. No. TR-388, U. of Maryland, College Park (August 1975).

[Howd 76] W.E. Howden, "Reliability of path analysis strategies", IEEE Trans. Soft. Eng. SE-2, 3 (1976) 208-214.

[Howd 80] W.E. Howden, "Functional program testing", IEEE Trans. Soft. Eng. SE-6, 2 (1980) 162-169.

[Howd 82] W.E. Howden, "Weak mutation testing and completeness of test sets", IEEE Trans. Soft. Eng. SE-8, 4 (1982) 371-379.

[HS 85] J. Hsiang and M. Srivas, "A Prolog environment for developing and reasoning about data types", Proc. Coll. Soft. Eng., Berlin (1985), this volume.

[Komo 80] H.J. Komorowski, "Qlog - The software for prolog and logic programming", Proc. of the Logic Programming Workshop, Debrecen, Hungary (1980) 305-320.

[Nais 82] L. Naish, "An introduction to MU-PROLOG", Technical Rept., Dept. of Computer Science, U. of Melbourne (1982).

[Paul 83] J. Paul, "Approche pour une certification fonctionnelle de systèmes à partir de spécifications externes", Internal rept., CIT, Lannion, France (1983).

[SW 83] D. Sanella and M. Wirsing, "A kernel language for algebraic specification and implementation", Proc. Int. Conf. Foundations Computing Theory, Bergholm, Sweden (Aug. 1983).

[WC 80] L.J. White, E.J. Cohen, "A domain strategy for computer program testing", IEEE Trans. Soft. Eng. SE-6, 3 (1980) 247-257.

[WHH 80] M.R. Woodward, D. Hedley and M.A. Hennel, "Experience with path analysis and testing of programs", IEEE Trans. Soft. Eng. SE-6, 3 (1980) 278-285.

[WPP 83] M. Wirsing, P. Pepper, H. Partsch, W. Dosch and M. Broy, "On hierarchies of abstract data types", Acta Informatica 20, 1 (Oct. 1983).

[ZW 81] S.H. Zeil, L.J. White, "Sufficient test sets for path analysis testing strategy", Proc. 5th Int. Conf. Soft. Eng., San Diego, Calif. (March 1981) 184-191.

APPENDIX

EXAMPLE: SORTED LISTS

The sort of interest is the sort sorted-list.
The lower sorts are the integer and boolean sorts.

Specification of the type list of integer, with the operation sort:

```
specif sorted-list =
 enrich bool, int by
  sort list;
  operations
          el :                    -> list     /* empty-list constructor */
          ap :        list * int  -> list     /* append constructor */
          sorted :    list        -> bool
          insert :    list * int  -> list     /* defined for a sorted list */
  variables
    L : list;
    X, Y : int;
  preconditions
/* The operation insert is used to insert an integer in a sorted list and to get as a
result a sorted list. */
    pre(insert,L,X) = (sorted(L) = true)
  axioms
    A1: sorted(el)=true
    A2: sorted(ap(el,X))=true
    A3: le(X,Y)=true  => sorted(ap(ap(L,X),Y))=sorted(ap(L,X))
    A4: le(X,Y)=false => sorted(ap(ap(L,X),Y))=false
    A5: insert(el,X)=ap(el,X)
    A6: le(X,Y)=true  => insert(ap(L,X),Y)=ap(ap(L,X),Y)
    A7: le(X,Y)=false => insert(ap(L,X),Y)=ap(insert(L,Y),X)
```

Specification of the integer type:

```
specif integer =
 enrich bool by
  sort int;
  operations
                0 :                 -> int      /* constructor */
                succ :    int       -> int      /* constructor */
                le :      int * int -> bool
  variables
    X,Y : int;
  axioms
    A8: le(0,X) = true
    A9: le(succ(X),0) = false
    A10: le(succ(X),succ(Y)) = le(X,Y)
```

Translation of the specification of the integer type into PROLOG:

C8: le(0,X,true).
C9: le(succ(X),0,false).
C10: le(succ(X),succ(Y),B):- le(X,Y,B).

Translation of the specification of the sorted-list type into PROLOG:

C1: sorted(el,true).
C2: sorted(ap(el,X), true).
C3: sorted(ap(ap(L,X),Y),B):- le(X,Y,true), sorted(ap(L,X),B).
C4: sorted(ap(ap(L,X),Y),false):- le(X,Y,false).
C5: insert(el,X,ap(el,X)):- sorted(el,true).
C6: insert(ap(L,X),Y,ap(ap(L,X),Y)):- sorted(ap(L,X),true),le(X,Y,true).
C7: insert(ap(L,X),Y,ap(Z,X)):- sorted(ap(L,X),true),le(X,Y,false),insert(L,Y,Z).

Instantiation sets generated for sorted:

We suppose that integer and boolean sorts are tested.
* For A1, the instantiation sets generated are empty for any n because there is no variable in this axiom: $\mathbf{I_n} = \{\}$, Vn
Thus an acceptable test sets collection is: $T_n = \{(sorted(el)=true)\}$, Vn

* For A2, there is no constraint on X. We make a uniformity hypothesis on integer and obtain the instantiation sets:
$\mathbf{I_n} = \{\mathbf{<meta\text{-}int>}\}$, Vn
$T_n = \{(sorted(ap(el,X))=true), X \in \mathbf{I_n}\}$, Vn

* For A3, the instantiation sets are made of tuples <L,X,Y>. There is a constraint on X,Y: le(X,Y,true), solved in PROLOG with the goal ?-*le(X,Y,true)*.
PROLOG answers:

$X_1 = 0$, $Y_1 = \_$;
$X_2 = succ(0)$, $Y_2 = succ(\_)$;
...
$X_{n+1} = succ^n(0)$, $Y_{n+1} = succ^n(\_)$;

As there is no constraint on the variable L of list sort, we make a uniformity hypothesis and substitute a meta-constant for the variable of this sort. Thus we deduce for a level n the instantiation set:

$\mathbf{I_n} = \{\mathbf{<meta\text{-}list_1, 0, meta\text{-}int_1>},$
$\mathbf{<meta\text{-}list_2, succ(0), succ(meta\text{-}int_2>},$
...
$\mathbf{<meta\text{-}list_n, succ^{n-1}(0), succ^{n-1}(meta\text{-}int_n)>}\}$
$T_n = \{(sorted(ap(ap(L,X),Y))=sorted(ap(L,X))), <L,X,Y> \in \mathbf{I_n}\}$

* For A4, the instantiation sets are obtained in a similar way and we get for a level n:

$\mathbf{I_n} = \{\mathbf{<meta\text{-}list_1, succ(meta\text{-}int_1), 0>},$
$\mathbf{<meta\text{-}list_2, succ(succ(meta\text{-}int_2)), succ(0)>},$
...
$\mathbf{<meta\text{-}list_n, succ^n(meta\text{-}int_n), succ^{n-1}(0)>}\}$
$T_n = \{(sorted(ap(ap(L,X),Y))=false), <L,X,Y> \in \mathbf{I_n}\}$

Instantiation sets generated for insert:

* For A5, as the only variable in the axiom is X, we obtain:

$\mathbf{I_n}$ = {**<meta-int>**}, Vn

T_n = {(insert(el,X)=ap(el,X)), X ∈ $\mathbf{I_n}$ }, Vn

* For A6, an instantiation set is made of tuples: <L,X,Y> with the constraint le(X,Y,true) on X and Y, and with the constraint sorted(ap(L,X),true) on L and X.
These constraints are solved with the goal: ?- *sorted(ap(L,X),true),le(X,Y,true).*
PROLOG answers:

L = el, X = 0, Y = _;
L = el, X = succ(0); Y = succ(_);
...

These answers are infinite and L is always equal to el: we are in an infinite branch. With a standard PROLOG interpreter, we obtain the following instantiation set for level n:

$\mathbf{I_n}$ = {**<el,0,meta-int$_1$>,**
<el,succ(0), succ(meta-int$_2$)>,
...
<el,succ^{n-1}(0), succ^{n-1}(meta-int$_n$)>}

T_n = {(insert(ap(L,X),Y) = ap(ap(L,X),Y)), <L,X,Y> ∈ $\mathbf{I_n}$ }

* For A7, the instantiation sets are obtained in a similar way with the goal:
?- *sorted(ap(L,X),true),le(X,Y,false).*
We obtain the following instantiation set for level n:

$\mathbf{I_n}$ = {**<el,succ(meta-int$_1$),0>,**
<el,succ(succ(meta-int$_2$)),succ(0)>,
...
<el,succ^n(meta-int$_n$),succ^{n-1}(0)>}

T_n = {(insert(ap(L,X),Y) = ap(insert(L,Y),X)), <L,X,Y> ∈ $\mathbf{I_n}$ }