RAPPORT DE RECHERCHE

L R I

TESTING DATA TYPES IMPLEMENTATIONS FROM ALGEBRAIC SPECIFICATIONS

GAUDEL M C / LE GALL P

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud – LRI

02/2007

Rapport de Recherche N° 1468

CNRS – Université de Paris Sud
Centre d'Orsay
LABORATOIRE DE RECHERCHE EN INFORMATIQUE
Bâtiment 490
91405 ORSAY Cedex (France)

Testing Data Types Implementations from Algebraic Specifications

Marie-Claude Gaudel[1], Pascale Le Gall[2]

[1] Université de Paris-Sud 11, LRI CNRS UMR 8623,
Bat. 490, F-91405 Orsay Cedex, France
mcg@lri.fr
[2]Université d'Évry-Val d'Essonne, IBISC CNRS FRE 2873,
523 pl. des Terrasses F-91025 Évry Cedex, France
pascale.legall@ibisc.univ-evry.fr

February 8, 2007

Abstract. Algebraic specifications of data types provide a natural basis for testing data types implementations. In this framework, the conformance relation is based on the satisfaction of axioms. This makes it possible to formally state the fundamental concepts of testing: exhaustive test set, testability hypotheses, oracle.

Various criteria for selecting finite test sets have been proposed. They depend on the form of the axioms, and on the possibilities of observation of the implementation under test. This last point is related to the well-known oracle problem. As the main interest of algebraic specifications is data type abstraction, testing a concrete implementation raises the issue of the gap between the abstract description and the concrete representation. The observational semantics of algebraic specifications bring solutions on the basis of the so-called observable contexts.

After a description of testing methods based on algebraic specifications, the chapter gives a brief presentation of some tools and case studies, and presents some applications to other formal methods involving datatypes.

Keywords: Specification-based testing, algebraic specifications, testability hypotheses, regularity and uniformity hypotheses, observability

1 Introduction

Deriving test cases from some descriptions of the Implementation Under Test (the IUT) is a very old and popular idea. In their pioneering paper [36], Goodenough and Gerhart pointed out that the choice of test cases should be based both on code coverage, and on specifications expressed by condition tables. One of the first papers where software testing was based on some formal description of the system under test, was by Chow [23]: software was modelled by finite state machines. It has been very influential on all the subsequent works on testing based on formal specifications.

Most approaches in this area are based on behavioural descriptions: for instance the control graph of the program, or some finite state machine or labelled

transition system. In such cases, it is rather natural to base the selection of test scenarios on some coverage criteria of the underlying graph.

Algebraic specifications are different: abstract data types are described in an axiomatic way [5, 14, 57]. There is a signature Σ, composed of a finite set S of sorts and a finite set F of function names over the sorts in S, and there is a finite set of axioms Ax. The correctness requirement is no more, as above, the ability (or the impossibility) for the IUT to exhibit certain behaviours: what is required by such specifications is the satisfaction of the axioms by the implementation of the functions of F. As a consequence, a natural way for testing some IUT is to choose some instantiations of the axioms (or of some consequences of them) and to check that when computed by the IUT, the terms occurring in the instantiations yield results that satisfy the corresponding axiom (or consequence). This approach was first proposed by Gannon et al. [33], and Bougé et al. [15, 16], and then developed and implemented by Bernot et al. [10].

Since these foundational works, testing from algebraic specifications has been investigated a lot. Numerous works have addressed different aspects.

Some authors as in [6] or [24] focus on a target programming language (Ada or Haskell). Testing from algebraic specifications has also been succesfully adapted for testing object-oriented systems [22, 31, 55]. Besides, methods inspired from algebraic testing have been applied to some other kinds of specifications like model-based specifications, first by Dick et al. [28], and more recently in [25]. Some other works explore links between test and proof [7, 18, 32].

Some tools [18, 24, 49] based either on resolution procedures or on specialised tactics in a proof engine, have been developed and used.

Extensions of algebraic specifications have also been studied, for instance, bounded datatypes [4] or partial functions [3]. More recently, some contributions [30, 46, 47] have been done to take into account structured or modular specifications aiming at defining structured test cases and at modelling the activity of both unit testing and integration testing.

Another special feature of algebraic specifications is the abstraction gap between the abstract specification level and the concrete implementation. This raises problems for interpreting the results of test experiments with respect to the specification. This characteristic is shared with other formal methods that allow the description of complex datatypes in an abstract way, for instance VDM, Z, and their object oriented extensions.

As a consequence, in the area of testing based on algebraic specifications, a special emphasis has been put on the oracle problem [3, 8, 43, 45, 58]. The oracle problem concerns the difficulty of defining reliable decision procedures to compare values of terms computed by the IUT. Actually, implementations of abstract data types may have subtle or complex representations, and the interface of the concrete datatypes is not systematically equipped with an equality procedure to compare values. In practice, only some basic datatypes provide a reliable decision procedure to compare values. They are said to be observable. The only way to define (partial) decision procedure for abstract data types is to observe them by applying some (composition of) functions yielding an observable result:

they are called observable contexts. Observational approaches of algebraic specifications bring solutions to define an appropriate notion of correctness taking into account observability issues.

The chapter is organised as follows: Section 2 presents some necessary basic notions of algebraic specifications; Section 3 gives the basic definitions of *test* and *test experiment* against an algebraic specification; Section 4 introduces in a progressive way the notions of *exhaustive test set* and *testability hypothesis* in a simple case. Then Section 5 addresses the issue of the selection of a finite test set via the so-called *uniformity* and *regularity* selection hypotheses. Section 6 develops further the theory, addressing the case where there are observability problems: this leads to a reformulation of the definitions mentioned above, and to a careful examination of the notion of correctness. Section 7 presents some of the most significant related pieces of work. The last section is devoted to brief presentations of some case studies, and to the descriptions of some transpositions of the framework to some other formal methods where it is possible to specify complex data types.

2 Preliminaries on algebraic specifications

Algebraic specifications of data types, sometimes called axiomatic specifications, provide a way of defining abstract data types by giving the properties (axioms) of their operations. There is no explicit definition of each operation (no pre- and post-condition, no algorithm) but a global set of properties that describes the relationship between the operations. This idea comes from the late seventies [35, 37]. It has been the origin of numerous pieces of work that have converged on the definition of CASL, the Common Algebraic Specification Language [14].

An example of an algebraic specification is given in Figure 1: it is a CASL specification of containers of natural numbers, i.e. a data structure that contains possibly duplicated numbers with no notion of order. This specification states that there are three sorts of values, namely Natural Numbers, Booleans and Containers. Among the operations, there is, for instance, a function named *isin* which, given two values, resp. of sort natural number and container, returns a boolean value. The operations must satisfy the axioms that are the formulas itemised by big bullets.

The sorts, operation names, and profiles of the operations are part of the *signature* of the specification. The signature gives the interface of the specified data type. Moreover, it declares some sorted variables that are used for writing the axioms.

An *(algebraic) signature* $\Sigma = (S, F, V)$ consists of a set S of sorts, a set F of function names each one equipped with an arity in $S^* \times S$ and a set of variables V, each of them being indexed by a sort. In the sequel, a function f with arity $(s_1 \dots s_n, s)$, where $s_1 \dots s_n, s \in S$, will be noted $f : s_1 \times \dots \times s_n \to s$.

In Figure 1, the sorts of the signature are Nat and $Bool$ (specified in some OUR/NUMBERS/WITH/BOOLS specification, not given here), and $Container$; the functions are [] (the empty container), _ :: _ (addition of a number to a

container), *isin* that checks for the belonging of a number to a container, and *remove* that takes away one occurrence of a number from a container; the variables are x, y of Nat sort, and c of $Container$ sort.

from OUR/NUMBERS/WITH/BOOLS **version** 0.0 **get** NAT, BOOL

spec CONTAINERS =
NAT, BOOL
then
generated type $Container ::= [] \mid \_::\_(Nat; Container)$
op $isin : Nat \times Container \to Bool$
op $remove : Nat \times Container \to Container$
$\forall\ x, y: Nat; c: Container$
• $isin(x, []) = \text{false}$ %(isin_empty)%
• $eq(x, y) = \text{true} \Rightarrow isin(x, y :: c) = \text{true}$ %(isin_1)%
• $eq(x, y) = \text{false} \Rightarrow isin(x, y :: c) = isin(x, c)$ %(isin_2)%
• $remove(x, []) = []$ %(remove_empty)%
• $eq(x, y) = \text{true} \Rightarrow remove(x, y :: c) = c$ %(remove_1)%
• $eq(x, y) = \text{false} \Rightarrow remove(x, y :: L) = y :: remove(x, c)$ %(remove_2)%
end

Fig. 1. An Algebraic specification of containers of natural numbers

Given a signature $\Sigma = (S, F, V)$, $T_\Sigma(V)$ is the set of *terms with variables in* V freely generated from variables and functions in Σ and preserving arity of functions. Such terms are indexed by the sort of their result. We note $T_\Sigma(V)_s$ the subset of $T_\Sigma(V)$ containing exactly those terms indexed by s.

T_Σ is the set $T_\Sigma(\emptyset)$ of the *ground terms* and we note $T_{\Sigma,s}$ the set of ground terms of sort s.

Considering the CONTAINER specification, an example of a ground term t of $Container$ sort is $0 :: 0 :: []$. An example of a term t' with variables is $isin(x, 0 :: c)$ that is of $Bool$ sort.

A *substitution* is any mapping $\rho : V \to T_\Sigma(V)$ that preserves sorts. Substitutions are naturally extended to terms with variables. The result of the application of a substitution ρ to a term t is called an *instantiation* of t, and is noted $t\rho$. In the example, let us consider the substitution $\sigma : \{x \to 0, y \to 0, c \to y :: []\}$, the instantiation $t'\sigma$ is the term with variable $isin(0, 0 :: y :: [])$.

Σ*-equations* are formulae of the form $t = t'$ with $t, t' \in T_\Sigma(V)_s$ for $s \in S$. An example of an equation on containers is $remove(x, []) = []$.

A *positive conditional* Σ*-formula* is any sentence of the form $\alpha_1 \wedge \ldots \wedge \alpha_n \Rightarrow \alpha_{n+1}$ where each α_i is a Σ-equation $(1 \leq i \leq n+1)$. $Sen(\Sigma)$ is the set of all positive conditional Σ-formulae.

A *(positive conditional) specification* $SP = (\Sigma, Ax, C)$ consists of a signature Σ, a set Ax of positive conditional formulae often called *axioms*, and some

constraints C, which may restrict the interpretations of the declared symbols (some examples are given below). When C is empty, we note $SP = (\Sigma, Ax)$ instead of $SP = (\Sigma, Ax, \emptyset)$.

Specifications can be structured as seen in the example: a specification SP can use some other specifications $SP_1, \ldots, SP_n$. In such cases, the signature is the union of signatures, and there are some *hierarchical* constraints that require the semantics of the used specifications to be preserved (for more explanations see [57]).

In the CONTAINERS specification, there are six axioms, named *isin_empty*, *isin_1*, *isin_2*, *remove_empty*, *remove_1*, and *remove_2*, and there is a so-called *generation* constraint, expressed at the line beginning by **generated type**, that all the containers are computable by composition of the functions [] and _ :: _. Such constraints are also called *reachability constraints*. The functions [] and _ :: _ are called the *constructors* of the *Container* type.

In some algebraic specification languages, axioms can be formulae of first-order logic, as in CASL. However, in this chapter we mainly consider positive conditional specifications[1].

A Σ-*algebra* $\mathcal{A}$ is a family of sets A_s, each of them being indexed by a sort; these sets are equipped, for each $f : s_1 \times \ldots \times s_n \to s \in F$, with a mapping $f^{\mathcal{A}} : A_{s_1} \times \ldots \times A_{s_n} \to A_s$. A Σ-morphism μ from a Σ-algebra $\mathcal{A}$ to a Σ-algebra $\mathcal{B}$ is a mapping $\mu : A \to B$ such that for all $s \in S$, $\mu(A_s) \subseteq B_s$ and for all $f : s_1 \times \ldots \times s_n \to s \in F$ and all $(a_1, \ldots, a_n) \in A_{s_1} \times \ldots \times A_{s_n}$ $\mu(f^{\mathcal{A}}(a_1, \ldots, a_n)) = f^{\mathcal{B}}(\mu(a_1), \ldots, \mu(a_n))$.

$Alg(\Sigma)$ is the class of all Σ-algebras.

Intuitively speaking, an implementation of a specification with signature Σ is a Σ-algebra: it means that it provides some sets of values named by the sorts, and some way of computing the functions on these values without side effect.

The set of ground terms T_Σ can be extended into a Σ-algebra by providing each function name $f : s_1 \times \ldots \times s_n \to s \in F$ with an application $f^{T_\Sigma} : (t_1, \ldots, t_n) \mapsto f(t_1, \ldots, t_n)$. In this case, the function names of the signature are simply interpreted as the syntactic constructions of the ground terms.

Given a Σ-algebra $\mathcal{A}$, we note $\_^A : T_\Sigma \to A$ the unique Σ-morphism that maps any $f(t_1, \ldots, t_n)$ to $f^{\mathcal{A}}(t_1^A, \ldots, t_n^A)$. A Σ-algebra $\mathcal{A}$ is said *reachable* if $\_^A$ is surjective.

A Σ-interpretation in A is any mapping $\iota : V \to A$. It is just an assignment of some values of the Σ-algebra to the variables. Given such an interpretation, it is extended to terms with variables: the value of the term is the result of its computation using the values of the variables and the relevant $f^{\mathcal{A}}$.

A Σ-algebra $\mathcal{A}$ *satisfies* a Σ-formula $\varphi : \wedge_{1 \leq i \leq n}\ t_i = t_i' \Rightarrow t = t'$, noted $\mathcal{A} \models \varphi$, if and only if for every Σ-interpretation ι in A, if for all i in $1..n$, $\iota(t_i) = \iota(t_i')$ then $\iota(t) = \iota(t')$. Given a specification $SP = (\Sigma, Ax, C)$, a Σ-algebra

[1] The reason is that most tools and case studies we present have been performed for and with this kind of specifications. An extension of our approach to first order logic, with some restrictions on quantifiers, was proposed by Machado in [46].

$\mathcal{A}$ is a *SP-algebra* if for every $\varphi \in Ax$, $\mathcal{A} \models \varphi$ and $\mathcal{A}$ fulfils the C constraint. $Alg(SP)$ is the subclass of $Alg(\Sigma)$ exactly containing all the SP-algebras.

A Σ-formula φ is a *semantic consequence* of a specification $SP = (\Sigma, Ax)$, noted $SP \models \varphi$, if and only if for every SP-algebra $\mathcal{A}$, we have $\mathcal{A} \models \varphi$.

3 Testing against an algebraic specification

Let SP be a positive conditional specification and IUT be an Implementation Under Test. In dynamic testing, we are interested in the properties of the computations by IUT of the functions specified in SP. IUT provides some procedures or methods for executing these functions. The question is whether they satisfy the axioms of SP.

Given a ground Σ-term t, we note t^{IUT} the result of its computation by IUT. Now we define how to test IUT against a Σ-equation.

Definition 1 (: Test and Test Experiment). *Given a Σ-equation ϵ, and IUT which provides an implementation for every function name of Σ,*

- *a test for ϵ is any ground instantiation $t = t'$ of ϵ;*
- *a test experiment of IUT against $t = t'$ consists in the evaluation of t^{IUT} and t'^{IUT} and the comparison of the resulting values.*

Example 1. One test of the *isin_empty* equation in the CONTAINERS specification of Figure 1 is $isin(0, []) = false$.

The generalization of this definition to positive conditional axioms is straightforward.

In the following, we say that a test experiment is successful if it concludes to the satisfaction of the test by the IUT, and we note it IUT *passes* τ where τ is the test, i.e. a ground formula. We generalise this notation to test sets: IUT *passes* TS means that $\forall \tau \in TS$, IUT *passes* τ.

Deciding whether IUT *passes* τ is the oracle problem mentioned in the introduction. In the above example it is just a comparison between two boolean values. However, such a comparison may be difficult when the results to be compared have complex data types. We postpone the discussion on the way it can be realised in such cases to Section 6. Actually, we temporarily consider in the two following sections that this decision is possible for all sorts, i.e. they are all "observable".

Remark 1. Strictly speaking, the definition above defines a tester rather than a test data: a test $t = t'$ is nothing else than the abstract definition of a program that evaluates t and t' via the relevant calls to the IUT and compares the results; a test experiment is an execution of this tester linked to the IUT.

We can now introduce a first definition of an exhaustive test of an implementation against an algebraic specification. A natural notion of correctness, when all the data types of the specification are observable, is that the IUT satisfies the axioms of the specification. Thus we start with a first notion of exhaustive test inspired from the notion of satisfaction as defined in Section 2.

4 A first presentation of exhaustivity and testability

Definition 2 (: Exhaustive Test Set, first version). *Given a positive conditional specification $SP = (\Sigma, Ax)$, an exhaustive test set for SP, noted $Exhaust_{SP}$, is the set of all well-sorted ground instantiations of the axioms in Ax:*

$$Exhaust_{SP} = \{\phi\rho \mid \phi \in Ax, \rho \in V \to T_\Sigma\}$$

An exhaustive test experiment of some IUT against SP is the set of all the test experiments of the IUT against the formulas of $Exhaust_{SP}$.

As said above, this definition is very close to (and is derived from) the notion of satisfaction of a set of Σ-axioms by a Σ-algebra. In particular, the fact that each axiom can be tested independently comes from this notion.

However, an implementation's passing once all the tests in the exhaustive test set does not necessarily mean that it satisfies the specification: first, this is true only if the IUT is deterministic; second, considering all the well-sorted ground instantiations is, a priori, not the same thing as considering all the Σ-interpretations in the values of the IUT. It may be the case that some values are not expressible by ground terms of the specification.

In other words, the above test set is exhaustive with respect to the specification, but may be not with respect to the values used by the program. Thus some *testability hypotheses* on the implementation under test are necessary: the success of the exhaustive test set ensures the satisfaction of the specification by the implementation only if this implementation behaves as a reachable Σ-algebra (cf. Section 2).

Practically, it means that:

- There is a realisation of every function of Σ that is supposed to be deterministic; the results do not depend on some hidden, non specified, internal state.
- The implementation is assumed to be developed following good programming practices; any computed value of a data type must always be a result of the specified operations of this data type.
- There is a comparison procedure for the values of every sort of the signature.

Note that, explicitly or not, all testing methods make assumptions on IUT: a totally erratic system, or a diabolic one, may pass some test set and fail later on[2]. In our case these hypotheses are static properties of the program. Some of them are (or could be) checkable by some preliminary static analysis of the source code.

[2] Testing methods based on Finite State Machine descriptions rely on the assumption that the IUT behaves as a FSM with the same number of states as the specification; similarly, methods based on IO-automata or IO-Transition Systems assume that the IUT behaves as an IO-automata: consequently, it is supposed input-enabled , i.e. always ready to accept any input.

Definition 3 (: Σ-Testability). *Given a signature Σ, an IUT is Σ-testable if it defines a reachable Σ-algebra $\mathcal{A}_{IUT}$. Moreover, for each τ of the form $t = t'$, there exists a way of deciding whether it passes or not.*

The Σ-testability of the IUT is called the minimal hypothesis H_{min} on the IUT.

Let us note $Correct(IUT, SP)$ the correctness property that a given IUT behaves as a reachable SP-algebra (i. e. the axioms are satisfied and all the values are specified). The fundamental link between exhaustivity and testability is given by the following formula:

$$H_{min}(IUT) \Rightarrow (\forall \tau \in Exhaust_{SP}, IUT \text{ passes } \tau \Leftrightarrow Correct(IUT, SP))$$

$Exhaust_{SP}$ is obviously not usable in practice since it is generally infinite. Actually, the aim of the definitions of $Exhaust_{SP}$ and H_{min} is to provide frameworks for developing theories of black-box testing from algebraic specifications. Practical test criteria (i.e. those which correspond to finite test sets) will be described as stronger hypotheses on the implementation. This point is developed in Sections 5 and 6.

Before addressing the issue of the selection of finite test sets, let us come back to the definition of $Exhaust_{SP}$. As it is defined, it may contain useless tests, namely those instantiations of conditional axioms where the premises are false: such tests are always successful, independently of the fact that their conclusion is satisfied by the IUT or not. Thus they can be removed.

Example 2. Assuming that $eq(0, 0) = true$ is a semantic consequence of the Our/Numbers/with/Bools specification, we can derive an equational test for the $remove\_1$ conditional axiom in the Containers specification of Figure 1. This test is simply the ground equation:
$remove(0, 0 :: 0 :: []) = 0 :: []$.

In the example of Figure 1, we have distinguished a subset of functions as constructors of the *Container* type (namely [] and ::). Under some conditions, the presence of constructors in a specification makes it possible to characterise an equational exhaustive test set.

A *signature with constructors* is a signature $\Sigma =< S, F, V >$ such that a subset $\mathcal{C}$ of elements of F are distinguished as constructors. Let us note $\Omega =< S, \mathcal{C}, V >$ the corresponding sub-signature of Σ, and T_Ω the corresponding ground terms. A specification $SP =< \Sigma, Ax >$ where Σ is a signature with constructors $\mathcal{C}$ is *complete with respect to its constructors* if and only if both following conditions hold:

- $\forall t \in T_\Sigma, \exists t' \in T_\Omega$ such that $SP \models t = t'$
- $\forall t, t' \in T_\Omega, SP \models t = t' \Rightarrow < \Sigma, \emptyset > \models t = t'$, i.e. t and t' are syntactically identical

Example 3. The Containers specification of Figure 1 is complete with respect to the constructors $\mathcal{C} = \{[], ::\}$ of the *Container* sort: from the axioms, any

ground term of *Container* sort containing some occurrence of the (non constructor) *remove* function is equal to some ground term containing only occurrences of [] and ::. Moreover, there is only one such ground term.

For such specifications and under some new hypotheses on the IUT, it is possible to demonstrate that the set of ground conclusions of the axioms is exhaustive. When removing premises satisfied by the specification, we should be careful not to remove some other premises that the IUT could interpret as true, even if they are not consequences of the specification. A sufficient condition is to suppose that the IUT correctly implements the constructors of all the sorts occurring in the premises. Let us introduce the new testability hypothesis $H_{min,\mathcal{C}}$ for that purpose. Intuitively, $H_{min,\mathcal{C}}$ means that the IUT implements data types with a syntax very close to their abstract denotation. It may seem to be a strong hypothesis, but in fact, it only applies to basic types, often those provided by the implementation language. As soon as the data type implementation is subtle or complex, the data type is then encapsulated and thus considered as non observable for testing (cf. Section 6).

Definition 4. *IUT satisfies $H_{min,\mathcal{C}}$ iff IUT satisfies H_{min} and :*

$$\forall s \in S, \forall u, v \in T_{\Omega,s},\ IUT\ passes\ u = v \Leftrightarrow SP \models u = v$$

Definition 5.

$$EqExhaust_{SP,\mathcal{C}} = \{\ \epsilon\rho \mid \exists \alpha_1 \wedge \ldots \wedge \alpha_n \Rightarrow \epsilon \in Ax, \\ \rho \in V \rightarrow T_\Omega, SP \models (\alpha_1 \wedge \ldots \wedge \alpha_n)\rho\}$$

Under $H_{min,\mathcal{C}}$ and for specifications complete with respect to their constructors $EqExhaust_{SP,\mathcal{C}}$ is an exhaustive test set. A proof can be found in [42] or in [1]. Its advantage over $Exhaust_{SP}$ is that it is made of equations. Thus the test experiments are simpler.

Some other approaches for the definitions of exhaustivity and testability are possible. For instance, as suggested in [11] and applied by Dong and Frankl in the ASTOOT system [31], a different possibility is to consider the algebraic specification as a term rewriting system, following a "normal-form" operational semantics. Under the condition that the specification defines a ground-convergent rewriting system, it leads to an alternative definition of the exhaustive test set:

$$Exhaust'_{SP} = \{t = t\downarrow \mid t \in T_\Sigma\}$$

where $t\downarrow$ is the unique normal form of t. The testability hypothesis can be weakened to the assumption that the IUT is deterministic (it does not need anymore to be reachable). In [31], an even bigger exhaustive test set was mentioned (but not used), which contained for every ground term the inequalities with other normal forms, strictly following the definition of initial semantics.

Actually, this is an example of a case where the exhaustive test set is not built from instantiations of the axioms, but more generally from an adequate set of semantic consequences of the specification. Other examples are shown in Section 6.

5 Selection hypotheses: uniformity, regularity

5.1 Introduction to selection hypotheses

A black-box testing strategy can be formalised as the selection of a finite subset of some exhaustive test set. In the sequel, we work with $EqExhaust_{SP,\mathcal{C}}$, but what we say is general to the numerous possible variants of exhaustive test sets.

Let us consider, for instance, the classical partition testing strategy[3]. It consists in defining a finite collection of (possibly non-disjoint) subsets that covers the exhaustive test set. Then one element of each subset is selected and submitted to the implementation under test. The choice of such a strategy corresponds to stronger hypotheses than H_{min} on the implementation under test. We call such hypotheses *selection hypotheses*. In the case of partition testing, they are called *uniformity hypothesis*, since the implementation under test is assumed to uniformly behave on some test subsets UTS_i (as Uniformity Test Subset):

$$UTS_1 \cup \ldots \cup UTS_p = EqExhaust_{SP,\mathcal{C}}, \text{ and}$$

$$\forall i = 1, \ldots, p, (\forall \tau \in UTS_i, IUT \ passes \ \tau \Rightarrow IUT \ passes \ UTS_i)$$

Various selection hypotheses can be formulated and combined depending on some knowledge of the program, some coverage criteria of the specification and ultimately cost considerations. Another type of selection hypothesis is *regularity hypothesis*, which uses a size function on the tests and has the form "if the subset of $EqExhaust_{SP,\mathcal{C}}$ made up of all the tests of size less than or equal to a given limit is passed, then $EqExhaust_{SP,\mathcal{C}}$ also is"[4].

All these hypotheses are important from a theoretical point of view because they formalise common test practices and express the gap between the success of a test strategy and correctness. They are also important in practice because exposing them makes clear the assumptions made on the implementation. Thus, they give some indication of complementary verifications, as used by Tse et al. in [20]. Moreover, as pointed out by Hierons in [40], they provide formal bases to express and compare test criteria and fault models.

5.2 How to choose selection hypotheses

As said above, the choice of the selection hypotheses may depend on many factors. However, in the case of algebraic specifications, the text of the specification provides useful guidelines. These guidelines rely on coverage of the axioms and composition of the cases occurring in premise of the axioms via unfolding as stated first in [10], and extended recently in [1].

[3] more exactly, it should be called sub-domain testing strategy.

[4] As noticed by several authors, [31], [20], and from our own experience [53], such hypotheses must be used with care. It is often necessary to choose this limit taking in consideration some "white-box knowledge" on the implementation of the datatypes: array bounds, etc

We recall that axioms are of the form $\alpha_1 \wedge \ldots \wedge \alpha_n \Rightarrow \alpha_{n+1}$ where each α_i is a Σ-equation $t_i = t'_i$, $(1 \leq i \leq n+1)$.

From the definition of $EqExhaust_{SP,\mathcal{C}}$, a test of such an axiom is some $\alpha_{n+1}\rho$ where $\rho \in V \to T_\Sigma$ is a well-typed ground substitution of the variables of the axiom such that the premise of the axiom, instantiated by ρ, is true: it is a semantic consequence of the specification $(SP \models (\alpha_1 \wedge \ldots \wedge \alpha_n)\rho)$.

One natural basic testing strategy is to cover each axiom once, i. e. to choose for every axiom one adequate substitution ρ only. The corresponding uniformity hypothesis is

$\forall \rho \in V \to T_\Sigma$ *such that* $SP \models (\alpha_1 \wedge \ldots \wedge \alpha_n)\rho$, *IUT passes* $\alpha_{n+1}\rho \Rightarrow$ (*IUT passes* $\alpha_{n+1}\rho', \forall \rho' \in V \to T_\Sigma$ *such that* $SP \models (\alpha_1 \wedge \ldots \wedge \alpha_n)\rho'$)

It defines a so-called *uniformity sub-domain* for the variables of the axiom that is the set of ground Σ-terms characterised by $SP \models (\alpha_1 \wedge \ldots \wedge \alpha_n)$.

Example 4. In the example of Figure 1, covering the six axioms requires six tests, for instance the following six ground equations:

- $isin(0, []) = false$, with the whole Nat sort as uniformity sub-domain;
- $isin(1, 1 :: 2 :: []) = true$, with the pairs of Nat such that $eq(x, y) = true$ and the whole $Container$ sort as uniformity sub-domain;
- $isin(1, 0 :: 3 :: []) = true$, with the pairs of Nat such that $eq(x, y) = false$ and the whole $Container$ sort as uniformity sub-domain;
- $remove(1, []) = []$, with the Nat sort as uniformity sub-domain;
- $remove(0, 0 :: 1 :: []) = 1 :: []$, with the pairs of Nat such that $eq(x, y) = true$ and the $Container$ sort as uniformity sub-domain;
- $remove(1, 0 :: []) = 0 :: []$, with the pairs of Nat such that $eq(x, y) = false$ and the $Container$ sort as uniformity sub-domain.

Such uniformity hypotheses are often too strong. A method for weakening them, and getting more test cases, is to compose the cases occurring in the axioms. In the full general case, it may involve tricky pattern matching on the premises and conclusions, and even some theorem proving. However, when the axioms are in a suitable form one can use the classical unfolding technique defined by Burstall and Darlington in [19]. It consists in replacing a function call by its definition. Thus, for unfolding to be applicable, the axioms must be organised as a set of functions definitions: every function is defined by a list of conditional equations such as:

$\wedge_{1 \leq i \leq m}\ \alpha_i \Rightarrow f(t_1, \ldots, t_n) = t$

where the domain of the function must be covered by the disjunction of the premises of the list.

Example 5. In the example of Figure 1, the *isin* function is defined by:

- $isin(x, []) = \text{false}$ %(isin_empty)%
- $eq(x, y) = \text{true} \Rightarrow isin(x, y :: c) = \text{true}$ %(isin_1)%
- $eq(x, y) = \text{false} \Rightarrow isin(x, y :: c) = isin(x, c)$ %(isin_2)%

It means that every occurrence of $isin(t_1, t_2)$ can correspond to the three following sub-cases:

- $t_2 = []$: in this case $isin(t_1, t_2)$ can be replaced by $false$;
- $t_2 = y :: c$ and $eq(t_1, y) = true$: in this case, it can be replaced by $true$;
- $t_2 = y :: c$ and $eq(t_1, y) = false$: in this case, it can be replaced by $y :: isin(t1, c)$.

A way of partitioning the uniformity sub-domain induced by the coverage of an axiom with some occurrence of $f(t_1, \ldots, t_n) = t$ is to introduce the sub-cases stated by the definition of f, and, of course, to perform the corresponding replacements in the conclusion equation to be tested. This leads to a weakening of the uniformity hypotheses.

Example 6. Let us consider the $isin\_2$ axiom. Its coverage corresponds to the uniformity sub-domain "pairs of Nat such that $eq(x, y) = false$" $\times$ "the $Container$ sort". Let us unfold in this axiom the second occurrence of $isin$, i.e. $isin(x, c)$. It leads to three sub-cases for this axiom:

- $c = []$:
  $eq(x, y) = false \wedge c = [] \Rightarrow isin(x, y :: []) = isin(x, [])$, i.e, false;
- $c = y' :: c'$ and $eq(x, y') = true$:
  $eq(x, y) = false \wedge c = y' :: c' \wedge eq(x, y') = true \Rightarrow isin(x, y :: y' :: c') = isin(x, y' :: c')$, i.e., true;
- $c = y' :: c'$ and $eq(x, y') = false$:
  $eq(x, y) = false \wedge c = y' :: c' \wedge eq(x, y') = false \Rightarrow isin(x, y :: y' :: c') = y :: isin(x, y' :: c')$, i.e. $isin(x, c')$.

The previous uniformity sub-domain is partitioned in three smaller sub-domains characterised by the three premises above. Covering these sub-cases leads to test bigger containers, and to check that $isin$ correctly behaves independently of the fact that the searched number was the last to be added to the container or not. Applying the same technique to the $remove\_2$ axiom leads to test that in case of duplicates, one occurrence only is removed.

Of course, unfolding can be iterated: the last case above can be decomposed again into three sub-cases. Unbounded unfolding leads generally to infinite test sets[5]. Limiting the number of unfoldings is generally sufficient for ensuring the finiteness of the test set. Experience has shown (see Section 8) that in practice one or two levels of unfolding are sufficient for ensuring what test engineers consider as a good coverage and a very good detection power. In some rare cases, this limitation of unfolding does not suffice for getting a finite test set: then, it must be combined with regularity hypotheses, i. e. limitation of the size of the ground instantiations.

[5] Actually, as it is described here, unbounded unfolding yields an infinite set of equations very close to the exhaustive test set. The only remaining variables are those that are operands of functions without definitions, namely, in our case, constructors

Unfolding has been implemented by Marre within the tool LOFT [10, 48, 49] using logic programming. There are some conditions on the specifications manipulated by LOFT:

- they must be complete with respect to constructors;
- when transforming the specification into a conditional rewriting system (by orienting each equation $t = t'$ occuring in an axiom from left to right $t \rightarrow t'$), the resulting conditional rewrite system must be confluent and terminating;
- each equation $t = t'$ that is the conclusion of an axiom must be such that t may be decomposed as a function f, not belonging to the set of constructors, applied to a tuple of terms built on constructors and variables only.

Under these conditions, the LOFT tool can decompose any uniformity domain into a family of uniformity sub-domains. It can also compute some solutions into a given uniformity sub-domain. These two steps correspond respectively to the computation of the uniformity hypotheses based on unfolding subdomains and to the generation of an arbitrary test case per each computed subdomain. The unfolding procedure is based on an equational resolution procedure involving some unification mechanisms. Under the conditions on the specifications given above, the unfolding procedure computes test cases such that: sub-domains are included in the domain they are issued from (soundness), and the decomposition into subdomains covers the splitted domain (completeness).

In [1], Aiguier et al. have extended the unfolding procedure for positive conditional specifications without restrictions. This procedure is also sound and complete. However, the price to pay is that instead of unfolding a unique occurrence of a defined function, the extended unfolding procedure requires to unfold all occurrences of the defined functions in a given equation among all the equations characterising the domain under decomposition. This may result in numerous test cases.

We have seen that conditional tests can be simplified into equational ones by solving their premises. It can be done in another way, replacing variables occurring in the axiom by terms as many times as necessary to find good instantiations. This method amounts to draw terms as long as the premises are not satisfied. This is particularly adapted in a probabilistic setting. In [9], Bouaziz et al. give some means to build some distributions on the sets of values.

6 Exhaustivity and testability versus observability

Until now, we have supposed that a test experiment $t = t'$ of the IUT may be successful or not depending on whether the evaluations of t and t' yield the same resulting values. Sometimes, comparing the test outputs may be a complex task when some information is missing. It often corresponds to complex abstract data types encapsulating some internal concrete data representations. Some abstract data types (sets, stacks, containers, etc) do not always provide an equality procedure within the implementation under test and we reasonnably cannot suppose

the existence of a finite procedure, the oracle, to correctly interpret the test results as equalities or inequalities. The so-called *oracle problem* in the framework of testing from algebraic specifications amounts to deal with equalities between terms of non observable sorts.

In this section, we distinguish a subset S_{Obs} of observable sorts among the set S of all sorts. For example, it may regroup all the sorts equipped with an equality predicate within the IUT environnement, for instance equality predicates provided by the programming language and considered as reliable. The minimal hypothesis H_{min} is relaxed to the weaker hypothesis H_{min}^{Obs} expressing that the the IUT still defines a reachable Σ-algebra but that the only remaining elementary tests which may be interpreted by the IUT as a verdict success/failure are the ground equality $t = t'$ of observable sort. The set Obs of all observable formulae is the subset of $Sen(\Sigma)$ of all formulae built over observable ground equalities. Any formula of Obs may be considered as a test experiment, and reciprocally.

The oracle problem in the case of non observable sorts may be tackled by two distinct but related questions. How to turn non observable equalities under test into test experiments tractable by an IUT only satisfying H_{min}^{Obs}? How far can we still talk about correctness when dealing with observability issues? Roughly speaking, the answers lie respectively in using observable contexts and in defining correctness up to some observability notion. We present these two corresponding key points in the following sections.

6.1 Observable contexts

In practice, non observable abstract data types can be observed through successive applications of functions leading to an observable result. It means that properties related to non observable sorts can be tested through observable contexts :

Definition 6 (: Context and Observable context).

An observable context c for a sort s is a term of observable sort with a unique occurrence of a special variable of sort s, generically denoted by z.

Such a context is often denoted $c[z]$ or simply $c[.]$ and $c[t]$ denotes $c\sigma$ where σ is the substitution associating the term t to the variable z.

An observable context is said to be minimal if it does contain an observable context as a strict subterm[6]*.*

Only minimal observable contexts are meaningful for testing. Indeed, if a context c has an observable context c' as a strict subterm, then $c[z]$ may be decomposed as $c_0[c'[z]]$. It implies that for any terms t and t', $c[t] = c[t']$ iff $c'[t] = c'[t']$. Both equalities being observable, the simpler one, $c'[t] = c'[t']$,

[6] A subterm of a term t is t itselt or any term occurring in it. In particular, if t is of form $f(t_1, \ldots, t_n)$ then $t_1, \ldots$ and t_n are subterms of t. A strict subterm of t is any subterm of t which differs from t.

suffices to infer whether $c[t] = c[t']$ holds or not. In the sequel, all the observable contexts will be considered as minimal by default.

For example, we can use set cardinality and element membership to observe some set data type as well as the height and the top of all successive popped stacks for some stack data type. Thus, a non observable ground equality of the form $t = t'$ is observed through all observable contexts $c[.]$ applied to both t and t'. From a testing point of view, it amounts to apply to both terms t and t' the same successive application of operations yielding an observable value, and to compare the resulting values.

Example 7. With the CONTAINERS specification of Figure 1, we now consider that the sort *Container* is no more an observable sort while Nat and $Bool$ are observable ones. Ground equalities of sort *Container* should be observed through the observable sorts Nat and $Bool$. An abstract test like $remove(3, []) = []$ is now observed through observable contexts. Each observable context of sort *Container* gives rise to a new (observable) test belonging by construction to Obs. For example, the context $isin(3, z)$ applied to the previous abstract test leads to the test : $isin(3, remove(3, [])) = isin(3, [])$.

In practice, there is often an infinity of such observable contexts. In the case of the CONTAINERS specification, we can build the following observable contexts[7]

$$isin(x, x_1 :: (x_2 :: \ldots (x_n :: z))),\ isin(x, remove(x_1, remove(x_2, \ldots, remove(x_n, z))))$$

or more generally, any combination of the operations *remove* and :: surrounded by the *isin* operation. As a consequence, we are facing a new kind of selection problem: to test an equality $t = t'$ of *Container* sort, one has to select among all these observable contexts a subset of finite or even reasonable size.

Bernot in [8] gives a counter-example based on the stack data type to assess that without additional information on the IUT, all the contexts are a priori necessary to test a non observable equality, even those involving constructors such as ::. More precisely, a context of the form $isin(x, x_1 :: z)$ may appear useless since it leads to build larger *Container* terms instead of observing the terms replacing z. In [8], it is shown that those contexts may reveal some programming errors depending on a bad use of state variables. From a theoretical point of view, let us consider a specification reduced to one axiom $a = b$ expressing that two non observable constants are equal. Then for any given arbitrary minimal context c_0, one can design a program P_{c_0} making $c[a] = c[b]$ true for all minimal observable contexts except c_0. This fact means that in general, any minimal context is needed to "fully" test non observable equalities. This is a simplified explanation of a proof given by Chen,Tse et al. in [20].

Let us point out that replacing an equation $t = t'$ by the (infinite) set of $c[t] = c[t']$ with c an observable context is classical within the community of algebraic specifications. Different observational approaches [13, 54] have been

[7] For convenience, we use the variables x, x_1, ..., x_n to denote arbitrary ground terms of sort Nat in a concise way.

proposed to cope with refinement of specifications based on abstract data types. They have introduced the so-called behavioural equalities, denoted by $t \approx t'$. The abstract equality is replaced by the (infinite) set of all observables contexts applying to both terms. More precisely, an algebra $\mathcal{A}$ satisfies $t \approx t'$ if and only if for every Σ-interpretations ι in A, for all observable contexts c, we have $\iota(c[t]) = \iota(c[t'])$. Behavioural equalities allow the specifier to refine abstract data types with concrete data types that do not satisfy some properties required at the abstract level. For example, the *Set* abstract data type with some axioms stating the commutativity of the element insertion, can be refined into the *List* abstract data type where the addition of an element by construction cannot be commutative. The refinement of *Set* by *List* is ensured by requiring that equalities on sets hold in the list specification only up to the behavioural equality. It amounts to state that observable operations (here the membership operation) behave in the same way at the abstract level of sets and at the implementation level of lists and to ignore those properties of the implementation that are not observable.

Considering an infinity of contexts is possible using context induction as defined by Hennicker in [39]. This is useful to prove a refinement step, but is useless in order to define an oracle. So, how can we select a finite set of observable contexts? Below we give some hints:

- The selection hypotheses presented in Section 5 to choose particular instantiations of axiom variables can be transposed to choose observable contexts. In particular, a rather natural way of selecting contexts consists in applying a regularity hypothesis. The size of a context is often defined in relation with the number of occurrences of non observable functions occurring in it.
- If one can characterise the equality predicate by means of a set of axioms, then one can use this axiomatisation, as proposed by Bidoit and Hennicker in [12], to define the test of non observable equalities. To give an intuition of how such an axiomatisation looks like, we give below the most classical one. It concerns the specification of abstract data types like sets, bags or containers, for which two terms are equal if and only if they exactly contain the same elements. Such an axiomatisation looks like:

  $$c \approx c' \text{ iff } \forall e, isin(e, c) = isin(e, c')$$

  where c and c' are variables of the abstract data type to be axiomatised, and e is a variable of element sort. $c \approx c'$ denotes the behavioural equality that is axiomatised. The axiomatisation simply expresses that the subset of contexts of the form $isin(e, z)$ suffices to characterize the behavioural equality. This particular subset of contexts can then be chosen as a suitable starting point to select observable contexts to test non observable equalities. Such an approach has two main drawbacks. First, such a finite aximatisation may not exist[8] or be difficult to guess. Second, selecting only from the subset

[8] For example, the classical stack specification has no finite axiomatisation of stack equality.

of observable contexts corresponding to a finite axiomatisation amounts to make an additional hypothesis on the IUT, which has been called the *oracle hypothesis* in [8]. In a few words, it consists in supposing that the IUT correctly implements the data type with respect to the functions involved in the axiomatisation. In the example of Containers, two containers are supposed to be behaviourally equal if and only if the membership operation *isin* applied on the containers always gives the same results. In other words, by using axiomatisation to build oracles, we are exactly supposing what we are supposed to test. Clearly, it may appear as a too strong hypothesis.

– Chen, Tse and others in [20] point out that some static analysis of the IUT may help to choose an adequate subset of observable contexts. When testing whether $t = t'$ holds or not, the authors compare their internal representations r and r' within the IUT. If r and r' are equal, then they can conclude[9] that the IUT passes $t = t'$. Otherwise, if r and r' are not equal, then they study which data representation components are different in r and r' and which are the observations which may reveal the difference. This makes it possible to build a subset of observable contexts which has a good chance to observationally distinguishes t and t'. The heuristic they have proposed has been successfully applied in an industrial context [56].

6.2 Correctness with observability issues

We have seen in Section 6.1 that the test of a non observable equality may be approached by a finite subset of observable contexts. More precisely, a non observable ground equality $t = t'$ may be partially verified by submitting a finite subset of the test set:

$$Obs(t = t') = \{c[t] = c[t'] \mid c \text{ is a minimal observable context}\}$$

.

The next question concerns testability issues : can we adapt the notions of correctness and exhaustivity when dealing with observability ? For example, one may wonder whether the set $Obs(t = t')$ may be considered as an exhaustive test set for testing the non observable (ground) equality $t = t'$. More generally, by taking inspiration from the presentation given in Section 4, we look for a general property linking the notions of exhaustive test set and testability such as:

$$H^{Obs}_{min}(IUT) \Rightarrow (\forall \tau \in Exhaust^{Obs}_{SP}, IUT \ passes \ \tau \Leftrightarrow Correct^{Obs}(IUT, SP))$$

6.2.1 Equational specifications

[9] [45] is partially based on this same idea : if the concrete implementations are identical, then necessarily their corresponding abstract denotations are equal terms.

If SP is an equational specification[10], then following Section 6.1, the test set

$$Exhaust_{SP}^{Obs} = \{\ c[t]\rho = c[t']\rho \mid t = t' \in Ax, \rho \in V \to T_\Sigma, \\ c \text{ minimal observable context}\}$$

is a good candidate[11] since it simply extends the $Obs(t = t')$ sets to the case of equations with variables. Actually, $Exhaust_{SP}^{Obs}$ is an exhaustive test set provided that we reconsider the definition of correctness taking into account observability.

By definition of observability, the IUT does not give access to any information on non observable sorts. Considering a given IUT as correct with respect to some specification SP should be defined up to all the possible observations and by discarding properties directly expressed on non observable sorts. Actually, observational correctness may be defined as : IUT is observationally correct with respect to SP according to the set of observations Obs, if there exists an SP-algebra $\mathcal{A}$ such that IUT and $\mathcal{A}$ exactly behave in the same way for all possible observations.

To illustrate, let us consider the case of the Container specification enriched by a new axiom of commutativity of element insertions:

$$x :: (y :: c) = y :: (x :: c)$$

The *Container* datatype is classically implemented by the *List* data type. However, elements in lists are usually stored according to the order of their insertion. In fact, the *List* data type is observationally equivalent to the *Container* data type as soon as the membership element is correctly implemented in the *List* specification. It is of little matter whether the *List* insertion function satisfies or not the axioms concerning the addition of elements in Containers.

This is formalised by introducing equivalence relations between algebras defined up to a set of Σ-formulae.

Definition 7. *Let $\Psi \subset Sen(\Sigma)$ and $\mathcal{A}$ and $\mathcal{B}$ be two Σ-algebras.*

$\mathcal{A}$ is said to be Ψ-equivalent to $\mathcal{B}$, denoted by $\mathcal{A} \equiv_\Psi \mathcal{B}$, if and only if we have $\forall \varphi \in \Psi, \mathcal{A} \models \varphi \iff \mathcal{B} \models \varphi$.

$\mathcal{A}$ is said to be observationally equivalent to $\mathcal{B}$ if and if $\mathcal{A} \equiv_{Obs} \mathcal{B}$.

We can now define observational correctness:

Definition 8. *Let IUT be an implementation under test satisfying H_{min}^{Obs}.*

IUT is observationally correct with respect to SP and according to Obs, denoted by $Correct^{Obs}(IUT, SP)$ if and only if

$$\exists \mathcal{A} \text{ reachable } SP\text{-algebra}, IUT \equiv_{Obs} \mathcal{A}$$

[10] Axioms of an equational specification are of the form $t = t'$ where t and t' are terms with variables and of the same sort.

[11] Let us remark that if t and t' are of observable sort s, then the only minimal observable context is z_s such that $t\rho = t'\rho$ are the unique tests associated to the axiom $t = t'$.

Remark 2. This notion of observational correctness has been first recommended for testing purpose by Le Gall and Arnould in [42, 43] for a large classe of specifications and observations[12]. With respect to the observational approaches in algebraic specifications [13], it corresponds to abstractor specifications for which the set of algebras is defined as the set of all algebras equivalent to at least an algebra of a kernel set, basically the set of all algebras satisfying the set of axioms.

From a testing point of view, each reachable SP-specification is obviously observationally correct with respect to SP. Reciprocally, an implementation IUT is observationally correct if it cannot be distinguished by observations from at least a reachable SP-algebra, say IUT_{SP}. So, nobody can say whether the implementation is the SP-algebra IUT_{SP}, and thus intrinsically correct, or the IUT is just an approximation of one reachable Σ-algebra up to the observations Obs. Thus, under the hypothesis H^{Obs}_{min}, any observationally correct IUT should be kept. Finally, $Correct^{Obs}(IUT, SP)$ captures exactly the set of all implementations which look like SP-algebras up to the observations in Obs. With this appropriate definition of $Correct^{Obs}(IUT, SP)$, the test set $Exhaust^{Obs}_{SP}$ is exhaustive. A sketch of the proof is the following. For each IUT passing $Exhaust^{Obs}_{SP}$, let us consider the quotient algebra $\mathcal{Q}$ built from IUT with the axioms of SP. We can then show that $\mathcal{Q}$ is a SP-algebra and is observationally equivalent to IUT.

6.2.2 Positive conditional specifications with observable premises

We also get an exhaustive test set when considering axioms with observable premises. For each axiom of the form $\alpha_1 \wedge \ldots \wedge \alpha_n \Rightarrow t = t'$ with all α_i of observable sort, it suffices to put in the corresponding exhaustive test set all the tests of the form $\alpha_1\rho \wedge \ldots \wedge \alpha_n\rho \Rightarrow c[t]\rho = c[t']\rho$ for all substitutions $\rho : V \rightarrow T_\Sigma$ and for all minimal observable contexts c.

Moreover, if we want to have an exhaustive test set involving equations only, as it has been done in Section 4, we should restrict to specifications with observable premises and complete with respect to the set $\mathcal{C}_{Obs}$ of constructors of observable sorts. As in Section 4, we also consider that the IUT correctly implements the constructors of all the sorts occurring in the premise, here the observable sorts[13]. That is to say, IUT satisfies $H_{min,\mathcal{C}_{Obs}}$ iff IUT satisfies H_{min} and :

$$\forall s \in S_{Obs}, \forall u, v \in T_{\Omega,s} IUT \; passes \; u = v \Leftrightarrow SP \models u = v$$

Under $H_{min,\mathcal{C}_{Obs}}$ and for the considered restricted class of specifications (i.e. observable premises and completeness with respect to $\mathcal{C}_{Obs}$),

[12] For interested readers, [10, 42, 43] give a generic presentation of formal testing from algebraic specifications in terms of institutions.

[13] When observable sorts coincide with the basic data types of the programming language, such an hypothesis is quite plausible. Thus, this is a weak hypothesis

$$EqExhaust_{SP}^{Obs} = \{c[t]\rho = c[t']\rho \mid \exists \alpha_1 \wedge \ldots \wedge \alpha_n \Rightarrow t = t' \in Ax, \rho \in V \to T_\Sigma, \\ c \text{ min. obs. context}, SP \models (\alpha_1 \wedge \ldots \wedge \alpha_n)\rho\}$$

is an exhaustive test set with respect to observational correctness.

6.2.3 Generalisation to non-observable premises

Is it possible to generalise such a construction of an exhaustive test set for specifications with positive conditional formulas comprising non-observable premises? A first naive solution would consist in replacing each non-observable equation $t = t'$ occuring either in the premise or in the conclusion of the axioms by a subset of $Obs(t = t')$. Unfortunately, such an idea cannot be applied, unless one accepts to submit biased tests[14]. This fact has been reported by Bernot and others in [8, 10]. To give an intuition, let us consider a new axiom

$$x :: x :: l = x :: l \Rightarrow true = false$$

which means that if addition to a container is idempotent, then[15] it would lead to $true = false$. Let us try to test the ground instance $0 :: 0 :: [] = 0 :: [] \Rightarrow true = false$ by considering a test ϕ in Obs of the form

$$\bigwedge_{\substack{\psi_i \in Obs(0 :: 0 :: [] = 0 :: []) \\ i \in I, I \text{ finite index}}} \psi_i \Rightarrow u = v$$

then the IUT may pass the premise

$$\bigwedge_{\substack{\psi_i \in Obs(0 :: 0 :: [] = 0 :: []) \\ i \in I, I \text{ finite index}}} \psi_i$$

without $0 :: 0 :: [] = 0 :: 0 :: []$ being a consequence of the specification. In that case, the IUT passes the test ϕ by passing the conclusion $true = false$. Thus, observing non observable premises through a finite set of contexts leads to require an observable equality, here $true = false$, which in fact is not required by the specification. This is clearly a bad idea.

It is now widely recognised that non-observable equations may be observed through some subset of observable contexts only when their position in the test is positive[16]. For example, the disjunctive normal form of $0 :: 0 :: [] = 0 :: [] \Rightarrow$

[14] A test is said to be biased when it rejects at least a correct implementation.

[15] This is no more than a positive conditional way of specifying $x :: x :: l \neq x :: l$. Actually, as the trivial algebra (with one element per sort) is satisfying all the conditional positive specifications, the inconsistency of specifications is often expressed by the possibility of deriving the boolean equation $true = false$.

[16] Roughly speaking, an atom $t = t'$ is said to be in a positive position if by putting the test into disjunctive normal form, then the $t = t'$ is not preceded by a negation.

$true = false$ is $\neg(0 :: 0 :: [] = 0 :: []) \vee true = false$ and thus $0 :: 0 :: [] = 0 :: []$ has a negative position in the test. In particular, Machado in [45, 46] considers any first order formula whose Skolem form does not contain existential quantifiers. Every non-observable equations in positive positions are observed by means of observable contexts while those in negative positions are observed by using concrete equality in the implementation. In that sense, Machado's approach is not a pure black-box approach deriving test cases and oracles from specifications but an approach mixing black-box and white-box where test cases are derived from the specifications and the oracle procedure is built from both the specification and the IUT.

We have shown that to deal with axioms with non-observable premises, it is not possible to apply observable contexts. However, can we do something else to handle such axioms? A tempting solution is to use the specification to recognise some ground instances of the axiom for which the specification requires the non observable premise to be true.

Let us come back to the axiom

$$x :: x :: l = x :: l \Rightarrow true = false$$

If it stands alone, nothing can be done to test it. Let us introduce a new axiom stating the idempotence law on the element insertion:

$$eq(x, y) = true \Rightarrow x :: y :: l = y :: l$$

Any ground instance of $x :: x :: l = x :: l$ is then a semantic consequence of the specification such that $true = false$ also becomes a semantic consequence. In such a case, one would like to consider $true = false$ as a test and even more, it seems rather crucial to precisely submit this test! This small example illustrates clearly why in this case, tests cannot be only ground instances of axioms but shoud be selected among all the observable semantic consequences of the specification[17] (see the end of Section 2 for the definition of semantic consequence.). Let us remark that according to the form of the specifications, one can use the unfolding technique described in Section 5 in order to solve the premise in the specification. In [42], Le Gall has shown that when the specification is complete with respect to the set $\mathcal{C}_{Obs}$ of constructors of observable sorts and under $H_{min,\mathcal{C}_{Obs}}$,

$$EqExhaust_{SP}^{Obs} = \{c[t]\rho = c[t']\rho \mid \exists \alpha_1 \wedge \ldots \wedge \alpha_n \Rightarrow t = t' \in Ax, \rho \in V \rightarrow T_\Sigma, \\ c \text{ min. obs. context}, SP \models (\alpha_1 \wedge \ldots \wedge \alpha_n)\rho\}$$

is an exhaustive test set with respect to observational correctness. Curiously, whether there are non-observable premises or not in the specification, the corresponding equational exhaustive test set is not modified.

[17] Observable semantic consequences are just those semantic consequences that belong to *Obs*. By construction, selecting a test outside this set would reject at least one correct implementation.

7 Related work

7.1 Related work on selection

In [24], Claessen and Hughes propose the QuickCheck tool for randomly testing Haskell programs from algebraic specifications. Axioms are encoded into executable Haskell programs whose arguments denote axiom variables. Conditional properties are tested by drawing data until finding a number, given as parameter, of cases which satisfy the premises. Of course, the procedure is stopped when a too large number of values is reached. The QuickCheck tool provides the user with test case generation functions for any arbitrary Haskell datatype, and in particular, also for functional types. The user can observe how the random data are distributed over the datatype carrier. When he considers that the distribution is not well balanced on the whole domain, for instance if the premises are satisfied by data of small size only, it is possible to specialise the test case generation functions to increase the likelihood to draw values ensuring a better coverage of the domain of premise satisfaction. This last feature is very useful for dealing with dependent datatypes. In [7], Berghofer and Nipkow use Quickcheck to exhibit counter-examples for universally quantified formulae written in executable Isabelle/HOL. This is a simple way to rapidly debug formalisation of a theory. In [32], Dydjer et al. develop a similar approach of using functional testing technics to help the proof construction by analysing counter-examples.

In [18], Brucker and Wolff use the full theorem proving environment Isabelle/HOL to present a method and a tool HOL-TestGen for generating test cases. They recommand to take benefit of the Isabelle/HOL proof engine equipped with tactics to transform a test domain (denoted as some proof goal) into test subdomains (denoted as proof subgoals). Selection hypotheses are expressed as proof hypotheses and the user can interact to guide the test data generation. Both the Quickcheck and TestGen tools present the advantage of offering an unified framework to deal with the specification, the selection and the generation of test cases, and even the submission of the test cases and the computation of the test verdict.

7.2 Related work on observability

We have given a brief account of observability considerations and their important impact on testability issues. In particular, there does not always exist an exhaustive test set, since such an existence depends on some properties of the specification and the implementation: namely, restrictions on the specification and hypotheses on the implementation.

The importance of observability issues for the oracle problem as been first raised by Bougé [16] and then Bernot, Gaudel and Marre in [8, 10]. It has been studied later on by Le Gall and Arnould [43] and Machado [3, 45]. Depending on the hypotheses on the possible observations and on the form or the extensions of the considered specifications, the oracle problem has been specialised. For example, in [4], Arnould et al. define a framework for testing from specifications

of bounded data types. To some extent, bounds of data types limit the possible observations : any data out of the scope of the bound description should not be observed when testing against such specifications. The set of observable formulae are formulae which are observable in the classical sense, where all terms are computed as being under the specified bound.

As soon as partial function are considered in the specification, it must be observable whether a term is defined or not. In [3], Arnould and Le Gall consider specifications with partial functions where definedness can be specified using an unary predicate def. The specification of equalities are declined with two predicates, strong equality $=$ allowing two undefined terms to be equal; existential equality $\overset{e}{=}$ for which only defined terms may be considered as equal. As the predicate $\overset{e}{=}$ may be expressed in term of $=$ and def, testing from specification with partiality naturally introduces two kinds of elementary tests directly related to the predicates def and $=$. Testing with partial functions requires to take into account the definition predicate: intuitively, testing whether a term is defined or not systematically precedes the following testing step, that is testing about equality of terms. Some initial results about testability and exhaustive test sets can be found in [3].

7.3 Variants of exhaustivity

Most exhaustive test sets presented here are made of tests directly derived from the axioms: tests are ground instances of (conclusions of) axioms, some equalities being possibly surrounded by observable contexts. Such tests do not necessarily reflect the practice of testing. Actually, the usual way of testing consists in applying the operation under test to some tuples of ground constructor terms and to compare the value computed by the IUT to a ground contructor term denoting the expected result. This can be described by tests of the form:

$$f(u_1, \ldots, u_n) = v$$

with f the function to be tested, and $u_1, \ldots, u_n, v$ ground constructor terms. The underlying intuition is that the constructor terms can denote all the concrete values manipulated by the implementation (reachability constraint). To illustrate this point of view, in the case of the CONTAINERS specification and by considering again that the sort $Container$ is observable, for the axiom $remove\_2$, instead of testing $remove(2, 3 :: []) = 3 :: remove(2, [])$ by solving the premise $eq(2, 3) = false$, a test of the good form would be $remove(2, 3 :: []) = 3 :: []$. Such a test may be obtained by applying the $remove\_1$ axiom to the occurrence $remove(2, [])$. In particular, LOFT [48, 49] computes tests of this reduced form. In [1, 3, 4], Arnould et al. present some exhaustive tests built from such tests involving constructor terms as much as possible.

7.4 The case of structured specifications

Until now, we have considered flat specifications which consist of a signature, a set of axioms, and possibly reachability constraints. Moreover, we have studied

the distinction between observable and non observable sorts. Observable sorts often correspond to the basic types provided by the programming environnement, and non observable sorts to the type of interest for the specification. However, algebraic specifications may be structured using various primitives allowing to import, combine, enrich, rename or forget (pieces of) imported specifications. Such constructions should be taken into account when testing.

As a first step to integration testing of systems described by structured algebraic specifications, Machado in [46, 47] shows how to build a test set whose structure is guided by the structure of the specification. The main and significant drawback of this approach is that hidden operations are ignored. As soon as an axiom involves an hidden operation, the axiom is not tested. Depending on the organisation of the specification, this can mean that a lot of properties are removed from the set of properties to be tested.

In [30], Doche and Wiels define a framework for composing test cases according to the structure of the specification. Their approach may be considered as modular since the IUT should have the same structure as the specification and the tests related to the subspecifications are composed together. These authors have established that correctness is preserved under some hypotheses[18] and have applied their approach to an industrial case study reported in [29].

8 Case studies and applications to other formal methods

This part of the paper briefly reports some case studies and experiments related to the theory presented here. Some of them were performed at LRI, some of them elsewhere. The first subsection is devoted to studies based on algebraic specifications. The next one reports interesting attempts to transpose some aspects of the theory to other formal approaches, namely VDM, Lustre, extended state machines and labelled transition systems. A special subsection presents some applications to object-oriented descriptions.

8.1 First case studies with algebraic specifications

A first experiment, performed at LRI by Dauchy and Marre, was on the on-board part of the driving system of an automatic subway[19] in collaboration with a certification agency. An algebraic specification was written [27]. Then two critical modules of the specification were used for experiments with LOFT: the overspeed controller and the door opening controller. These two modules shared the use of eight other specification modules that described the state of the on-board system. The number of axioms for the door controller was 25, with rather complex premisses. The number of axioms of the speed controller was 34. There where 108 function names and several hundred axioms in the shared

[18] For interested readers, the hypotheses aim at preserving properties along signature morphisms and thus, are very close to the satisfaction condition of the institution framework.

[19] precisely, the train controller on line D in Lyon that has been operating since 1991.

modules. Different choices of uniformity hypotheses were experienced for the door controller: they led to 230, 95, and 47 tests. For the overspeed controller, only one choice was sensible and led to 95 tests. The experiment is reported in details in [26]. In a few words, these tests were used by the certification team as a sort of checklist against the tests performed by the development team. This approach led to the identification of a tricky combination of conditions that had not been tested by the developers.

A second experiment is reported in [53] and was performed within a collaboration between LRI and the LAAS laboratory in Toulouse. The experiment was performed on a rather small piece of software written in C, which was extracted from a nuclear safety shutdown system. The piece of software contained some already known bugs that were discovered but one: it was related to some hidden shared variable in the implementation, and required rather large instantiations, larger than the bound chosen a priori for the regularity hypothesis. On a theoretical point of view, this can be analysed as a case where the testability hypothesis was not ensured. More practically, the fault was easy to detect by "white-box" methods, either static analysis or structural testing with branch coverage. This is coherent with the remark in Section 4 on the possibility of static checking of the testability hypothesis, and with the footnote 4 in Section 5 on the difficulties to determine adequate bounds for regularity hypotheses.

An experiment of "intensive" testing of the EPFL library of Ada components was led by Buchs and Barbey in the Software Engineering Laboratory at EPFL [6]. First an algebraic specification of the component was reengineered: the signature was derived from the package specifications of the family, and the axioms were written manually. Then the LOFT system was used with a standard choice of hypotheses.

LOFT has been also used for the validation of a transit node algebraic specification [2]. Generating test cases was used for enumerating scenarios with a given pattern. It led to the identification of one undesirable, and unexpected, scenario in the formal specification.

It was also used for the test of the data types of an implementation of the Two-Phase-Commit protocol [34] without finding any fault: this was probably due to the fact that the implementation had been systematically derived from a formal specification. Other aspects of this case study are reported in the next subsection.

The specifications and test sets of these case studies are too large to be given here. Details can be found in [27] and [26] for the first one, in [2] and [50] for the transit node, and in [41] for the Two-Phase-Commit protocol.

8.2 Applications to other methods

Actually, the approach developed here for algebraic data types is rather generic and presents a general framework for test data selection from formal specifications. It has been reused for, or has inspired, several test generation methods from various specification formalisms: VDM, Lustre, full LOTOS.

The foundational paper by Jeremy Dick and Alain Faivre on test case generation from VDM specifications [28] makes numerous references to some of the notions and techniques presented here, namely uniformity and regularity hypotheses, and unfolding. The formulae of VDM specifications are relations on states decribed by operations (in the sense of VDM, i.e. state modifications). They are expressed in first-order predicate calculus. These relations are reduced to a disjunctive normal form (DNF), creating a set of disjoint sub-relations. Each sub-relation yields a set of constraints which describe a single test domain. The reduction to DNF is similar to axiom unfolding: uniformity and regularity hypotheses appear in relation with this partition analysis. As VDM is state-based, it is not enough to partition the operations domains. Thus the authors give a method of extracting a finite state automaton from a specification. This method uses the results of the partition analysis of the operations to perform a partition analysis of the states. This led to a set of disjoint classes of states, each of which corresponds either to a precondition or a postcondition of one of the above sub-relations. Thus, a finite state automaton can be defined, where the states are some equivalence classes of states of the specifications. From this automaton, some test suites are produced such that they ensure a certain coverage of the automaton paths. The notion of test suites is strongly related to the state orientation of the specification: it is necessary to test the state evolution in presence of sequences of data, the order being important.

Test generation from Lustre descriptions has been first studied jointly at CEA and LRI. The use of the LOFT system to assist the test of Lustre programs has been investigated. Lustre is a description language for reactive systems which is based on the synchronous approach [38]. An algebraic semantics of Lustre was stated and entered as a specification in LOFT. Lustre programs were considered as enrichments of this specification, just as some specific axiom to be tested. After this first experience, GATEL, a specific tool for Lustre was developed by Marre at CEA (Commissariat à l'Énergie Atomique). In GATEL, a Lustre specification of the IUT, and some Lustre descriptions of environment constraints and test purpose are interpreted via Constraint Logic Programming. Unfolding is the basic technique, coupled with a specific constraint solving library [51, 52]. GATEL is used at IRSN (Institut de Radioprotection et Sûreté Nucléaire) for identifying those reachable classes of tests covering a given specification, according to some required coverage criteria. The functional tests performed by the developers are then compared to these classes in order to point out uncovered classes, i.e. insufficient testing. If it is the case, GATEL provides test scenarios for the missing classes.

LOTOS is a well known formal specification language, mainly used in the area of communication protocols. There are two variants: basic LOTOS makes it possible to describe processes and their synchronisation, with no notion of data type; full LOTOS, where it is possible to specify algebraic data types and how their values can be communicated or shared between specified processes. In the first case, the underlying semantics of a basic LOTOS specification is a finite labelled transition system. There is an extremely rich corpus of testing methods

based on such finite models (see [17] for an annotated bibliography). However, there are few results on extending them to infinite models, as it is the case when non trivial data types are introduced. In [34], Gaudel and James have stated the underlying notions of testability hypotheses, exhaustive test sets, and selection hypotheses for full LOTOS.

This approach has been used by James for testing an implementation of the Two-Phase-Commit Protocol developed from a LOTOS specification into Concert/C. The results of this experiment are reported in [41].As said in the previous sub-section, tests for the data types were obtained first with the LOFT system. Then a set of testers was derived manually from the process part of the specification. The submission of these tests, was preceded by a test campaign of the implementations of the atomic actions of the specification by the Concert/C library, i. e. the communication infrastructure (the set of gates connecting the processes), which was developed step by step. It was motivated by the testability hypothesis: it was a way of ensuring the fact that the actions in the implementation were the same as in the specification, and that they were atomic. No errors were found in the data types implementations, but an undocumented error of the Concert/C pre-processor was detected when testing them. Some errors were discovered in the implementation of the main process. They were related to memory management, and to the treatment of the time-outs. There are always questions on the interest of testing pieces of software, which have been formally specified and almost directly derived from the specification. But this experiment shows that problems may arise: the first error-prone aspect, memory management, was not expressed in the LOTOS specification because of its abstract nature; the second one was specified in a tricky way due to the absence of explicit time in classical LOTOS. Such unspecified aspects are unavoidable when developing efficient implementation.

8.3 Applications to object-oriented software

It is well known that there is a strong relationship between abstract data types and object orientation. There is the same underlying idea of encapsulation of the concrete implementation of data types. Thus it is not surprising that the testing methods presented here for algebraic specifications has been adapted to the test of object oriented systems. We present two examples of such adaptations.

The ASTOOT approach was developped by Dong and Frankl at the Polytechnic University in New-York [31]. The addressed problem was the test of object-oriented programs: classes are tested against algebraic specifications. A set of tools had been developed. As mentioned at the end of Section 4, a different choice was made for the exhaustive test set, which is the set of equalities of every ground term with its normal form, and it was also suggested to test inequalities of ground terms As normal forms are central in the definition of tests, there was a requirement that the axioms of the specification must define a convergent term rewriting system. Moreover, there is a restriction to classes such that their operations have no side effects on their parameters and functions have no side effects: it corresponds to a notion of testability. The oracle problem was

addressed by introducing a notion of observational equivalence between objects of user-defined classes, which is based on minimal observational contexts, and by approximating it. Similarly to Section 5, the test case selection was guided by an analysis of the conditions occuring in the axioms; the result was a set of constraints that was solved manually. The theory presented here for algebraic data types turned out to nicely fit to cope with object-orientation, even when different basic choices were made.

This had been confirmed by further developments by Tse and its group at the university of Hong Kong [21, 22, 56]. In their approach, object-oriented systems are described by algebraic specifications for classes and contract specification for clusters of related classes : contracts specify interactions between objects via message-passing rules. As in our approach, some tests are fundamental pairs of equivalent ground terms obtained via instantiations of the axioms. As in ASTOOT non equivalent pairs of terms are also considered. Some white-box heuristic for selecting relevant observable contexts makes it possible to determine whether the objects resulting from executing such test cases are observationally equivalent. Moreover, message passing test sequences are derived from the contract specification and the source code of the methods. This method has been recently applied for testing object-oriented industrial software [56].

9 Conclusion

Algebraic specifications have proved to be an interesting basis for stating some theory of black-box testing and for developing methods and tools. The underlying ideas have turned out to be rather general and applicable to specification methods including datatypes, whatever the formalism used for their description. It is the case of the notions of uniformity hypothesis, and regularity hypotheses that have been reused in other contexts.

In presence of abstraction and encapsulation, the oracle problem raises difficult issues due to the limitations on the way concrete implementations can be observed and interpreted. This is not specific to algebraic specifications and abstract data types: the same problems arise for embedded and/or distributed systems. It is interesting to note the similarity between the observable contexts presented here, and the various ways of distinguishing and identifying the state reached after a test sequence in finite state machines [44], namely separating families, distinguishing sequences, characterising sets, and their variants.

The methodology presented here has been applied, as such or with some adjustments, in a significant number of academic or industrial case studies. In most cases, they have been used for some a posteriori certification of critical systems that had already been intensively validated and verified, or for testing implementations that have been developed from some formal specification. This is not surprising: in the first case, the risks are such that certification agencies are ready to explore sophisticated methods; in the second case, the availability of the formal specification pushes for using it for test generation. In both circumstances, it was rather unlikely to find errors. But some were discovered however, and

missing test cases were identified. In some cases, these detections were quite welcome and prevented serious problems. This is an indication of the interest of test methods based on formal specifications, and of the role they can play in the validation and verification process.

References

1. Marc Aiguier, Agnès Arnould, Clément Boin, Pascale Le Gall, and Bruno Marre. Testing from algebraic specifications: Test data set selection by unfolding axioms. In *Formal Approaches to Software Testing, 5th International Workshop, FATES 2005, Edinburgh, UK, July 11, 2005, Revised Selected Papers*, volume 3997 of *Lecture Notes in Computer Science*, pages 203–217. Springer, 2005.
2. A. Arnold, M. Gaudel, and B. Marre. An experiment on the validation of a specification by heterogeneous formal means: The transit node. In *5th IFIP Working Conference on Dependable Computing for Critical Applications (DCCA5)*, pages 24–34, 1995.
3. A. Arnould and P. Le Gall. Test de conformité: une approche algébrique. *Technique et Science Informatiques, Test de logiciel, vol. 21, n° 9*, pages 1219–1242, 2002.
4. A. Arnould, P. Le Gall, and B. Marre. Dynamic testing from bounded data type specifications. In *Dependable Computing - EDCC-2*, volume 1150 of *Lecture Notes in Computer Science*, pages 285–302, Taormina, Italy, Octobre 1996. Springer.
5. E. Astesiano, H.-J. Kreowski, and B. Krieg-Bruckner. *Algebraic Foundations of Systems Specification*. IFIP State-of-the-Art Reports. Springer-Verlag, 1999.
6. S. Barbey and D. Buchs. Testing Ada abstract data types using formal specifications. In *1st Int. Eurospace-Ada-Europe Symposium*, number 887 in Lecture Notes in Computer Science, pages 76–89. Springer Verlag, 1994.
7. S. Berghofer and T. Nipkow. Random testing in isabelle/hol. In *SEFM*, pages 230–239, 2004.
8. G. Bernot. Testing against formal specifications: a theoretical view. In *TAPSOFT'91, International Joint Conference on the Theory and Practice of Software Development*, volume 494 of *Lecture Notes in Computer Science*, pages 99–119. Springer Verlag, 1991.
9. G. Bernot, L. Bouaziz, and P. Le Gall. A theory of probabilistic functional testing. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 216–226. ACM Press, 1997.
10. G. Bernot, M.-C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387–405, 1991.
11. G. Bernot, M.-C. Gaudel, and B. Marre. A formal approach to software testing. In *2nd International Conference on Algebraic Methodology and Software Technology (AMAST)*, number 670 in Worshops in Computing Series, pages 243–253. Springer Verlag, 1992.
12. M. Bidoit and R. Hennicker. Behavioural theories and the proof of behavioural properties. *Theoretical Computer Science*, 165(1):3–55, 1996.
13. M. Bidoit, R. Hennicker, and M. Wirsing. Behavioural and abstractor specifications. *Science of Computer Programming*, 25(2-3):149–186, 1995.
14. M. Bidoit and P. D. Mosses. *CASL user manual*. Number 2900 in Lecture Notes in Computer Science. Springer-Verlag, 1998.
15. L. Bougé. Modélisation de la notion de test de programmes, application à la production de jeux de test. Ph. D. thesis, Université de Paris 6, 1982.

16. L. Bougé, N.Choquet, L. Fribourg, and M.-C. Gaudel. Test set generation from algebraic specifications using logic programming. *Journal of Systems and Software*, 6(4):343–360, 1986.
17. E. Brinksma and J. Tretmans. Testing transition systems: An annotated bibliography. In *MOVEP 2000 summer school*, volume 2067 of *Lecture Notes in Computer Science*, pages 187–195. Springer, 2001.
18. A. D. Brucker and B. Wolff. Symbolic test case generation for primitive recursive functions. In *Workshop on Formal Approaches to Testing of Software, 2004*, volume 3395 of *Lecture Notes in Computer Science*, pages 16–32. Springer Verlag, 2005.
19. R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, 1977.
20. H. Y. Chen, T. H. Tse, F. T. Chan, and T. Y. Chen. In black and white: an integrated approach to class-level testing of object-oriented programs. *ACM transactions on Software Engineering and Methodology*, 7(3):250–295, 1998.
21. H. Y. Chen, T. H. Tse, F. T. Chan, and T.Y. Chen. In black and white: an integrated approach to class-level testing of object-oriented programs. *ACM transactions on Software Engineering and Methodology*, 7(3):250–295, 1998.
22. H. Y. Chen, T. H. Tse, and T. Y. Chen. TACCLE: a methodology for object-oriented software testing at the class and cluster levels. *ACM Transactions on Software Engineering and Methodology*, 10(1):56–109, 2001.
23. T.S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, 1978.
24. K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *International Conference on Functional Programming*, pages 268–279, 2000.
25. L. Dan and B. K. Aichernig. Combining algebraic and model-based test case generation. In *ICTAC 04*, 2004.
26. P. Dauchy, M.-C. Gaudel, and B. Marre. Using algebraic specifications in software testing : a case study on the software of an automatic subway. *Journal of Systems and Software*, 21(3):229–244, 1993.
27. P. Dauchy and P. Ozello. Experiments with formal specifications on MAGGALY. In *Second International Conference on Applications of Advanced Technologies in Transportation Engineering*, Mineapolis, 1991.
28. J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Formal Methods Europe'93*, number 670 in Lecture Notes in Computer Science, pages 268–284. Springer Verlag, 1993.
29. M. Doche, C. Seguin, and V. Wiels. A modular approach to specify and test an electrical flight control system. In *FMICS-4*, 1999.
30. M. Doche and V. Wiels. Extended institutions for testing. In *AMAST'2000*, number 1816 in Lecture Notes in Computer Science, pages 514–528, 2000.
31. R. K. Dong and Ph. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):103–130, 1994.
32. P. Dybjer, Q. Haiyan, and M. Takeyama. Combining testing and proving in dependent type theory. In *TPHOLs 2003*, volume 2758 of *Lecture Notes in Computer Science*, pages 188–203, 2003.
33. J. Gannon, P. McMullin, and R. Hamlet. Data abstraction implementation, specification and testing. *ACM Transactions on on Programming Languages and Systems*, 3(3):211–223, 1981.
34. M.-C. Gaudel and P. J. James. Testing algebraic data types and processes : a unifying theory. *Formal Aspects of Computing*, 10(5-6):436–451, 1998.

35. J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In *Current Trends in Programming Methodology IV : Data structuring*, pages 80–144. Prentice Hall, 1978.
36. J. B. Goodenough and S. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1(2):156–173, 1975.
37. J.V. Guttag and J.J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10(1):27–52, 1978.
38. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
39. R. Hennicker. Context induction: a proof principle for behavioural abstractions and algebraic implementations. *Formal Aspects of Computing*, 3(4):326–345, 1991.
40. R. M. Hierons. Comparing test sets and criteria in the presence of test hypotheses and fault domains. *ACM Trans. Softw. Eng. Methodol.*, 11(4):427–448, 2002.
41. P. R. James, M. Endler, and M.-C. Gaudel. Development of an atomic broadcast protocol using LOTOS. *Software Practice and Experience*, 29(8):699–719, 1999.
42. P. Le Gall. *Les algèbres étiquetées : une sémantique pour les spécifications algébriques fondée sur une utilisation systématique des termes. Application au test de logiciel avec traitement d'exceptions.* PhD thesis, Université de Paris XI, Orsay, 1993.
43. P. Le Gall and A. Arnould. Formal specification and test: correctness and oracle. In *11th WADT joint with the 9th general COMPASS workshop*, volume 1130 of *Lecture Notes in Computer Science*, pages 342–358. Springer, 1996. Oslo, Norway, Sep. 1995, Selected papers.
44. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.
45. P. Machado. On oracles for interpreting test results against algebraic specifications. In A.M. Haeberer, editor, *Algebraic Methodology and Software Technology, AMAST'98*, volume 1548 of *Lecture Notes in Computer Science*, pages 502–518, 1998.
46. P. Machado. Testing from structured algebraic specifications. In *AMAST2000*, volume 1816 of *Lecture Notes in Computer Science*, pages 529–544, 2000.
47. P. Machado and D. Sannella. Unit testing for CASL architectural specifications. In *Mathematical Foundations of Computer Science*, number 2420 in Lecture Notes in Computer Science, pages 506–518. Springer-Verlag, 2002.
48. B. Marre. Toward an automatic test data set selection using algebraic specifications and logic programming. In K. Furukawa, editor, *Eight International Conference on Logic Programming (ICLP'91)*, pages 25–28. MIT Press, 1991.
49. B. Marre. Loft : a tool for assisting selection of test data sets from algebraic specifications. In *TAPSOFT*, volume 915 of *Lecture Notes in Computer Science*, pages 799–800. Springer Verlag, 1995.
50. B. Marre, A. Arnold, and M.C. Gaudel. Validation d'une spécification par des formalismes différents: le noeud de transit. *Revue Technique et Science Informatiques*, 16(6):677–699, 1997.
51. B. Marre and A. Arnould. Test sequences generation from LUSTRE descriptions : GATEL. In *15h I.E.E.E. International Conference on Automated Software Engineering*, pages 229–237, 2000.
52. B. Marre and B. Blanc. Test selection strategies for lustre descriptions in gatel. In *MBT 2004 joint to ETAPS'2004*, volume 111 of *ENTCS*, pages 93–111, 2004.

53. B. Marre, P. Thévenod-Fosse, H. Waeselink, P. Le Gall, and Y. Crouzet. An experimental evaluation of formal testing and statistical testing. In *SAFECOMP'92*, pages 311–316, 1992.
54. F. Orejas, M. Navarro, and A. Sanchez. Implementation and behavioural equivalence: A survey. In *Recent Trends in Data Type Specification, 8th Workshop on Specification of Abstract Data Types joint with the 3rd COMPASS Workshop*, volume 655 of *Lecture Notes in Computer Science*, pages 144–163. Springer Verlag, 1993.
55. C. Péraire, S. Barbey, and D. Buchs. Test selection for object-oriented software based on formal specifications. In *IFIP Working Conference on Programming Concepts and Methods (PROCOMET'98), Shelter Island, New York, USA, June 1998*, pages 385–403. Chapman Hall, 1998.
56. T.H. Tse, F. C. M. Lau, W.K. Chan, P. C. K. Liu, and C.K.F. Luk. Testing object-oriented industrial software without precise oracles or results. *Communications of the ACM*, accepted in 2006.
57. M. Wirsing. *Handbook of Theoretical Computer Science*, volume B, Formal models and semantics, chapter Algebraic Specification. Elsevier, 1990.
58. H. Zhu. A note on test oracles and semantics of algebraic specifications. In *QSIC 2003*, pages 91–99. IEEE Computer Society, 2003.