Testing can be formal, too

Marie-Claude Gaudel

LRI, URA 410 du CNRS
Université de Paris-Sud, Batiment 490
91405 Orsay, France

Abstract. The paper presents a theory of program testing based on formal specifications. The formal semantics of the specifications is the basis for a notion of an exhaustive test set. Under some minimal hypotheses on the program under test, the success of this test set is equivalent to the satisfaction of the specification.
The selection of a finite subset of the exhaustive test set can be seen as the introduction of more hypotheses on the program, called selection hypotheses. Several examples of commonly used selection hypotheses are presented.
Another problem is the observability of the results of a program with respect to its specification : contrary to some common belief, the use of a formal specification is not always sufficient to decide whether a test execution is a success. As soon as the specification deals with more abstract entities than the program, program results may appear in a form which is not obviously equivalent to the specificied results. A solution to this problem is proposed in the case of algebraic specifications.

1 Introduction

This paper is a survey of a research activity which has been led for several years in the area of program testing [5, 6, 7], and a presentation of some recent results [2, 3, 22].

There has been surprisingly little research on the foundations of program testing. Some notable exceptions are the early paper by Goodenough and Gerhart [18] where the notion of testing criteria was first introduced and the work of Gourlay [19] who proposed a formal definition of a test method and pointed out the role of specifications in the testing process. More recently, the growing interest in the use of formal methods has raised several works on testing based on formal specifications. It turns out that formal specifications are quite fundamental to a rigourous and systematic approach of program testing. For instance, an application area where testing has been extensively studied is the area of communication protocols [8, 15, 27], etc: all these works are based on formal models of the protocol to be tested, generally finite state machines or transition systems.

First, let us make more precise the background of the works reported here. The aim of a testing activity is to detect some faults in a program. The use of the verb *detect* is important here, since locating and correcting the faults are generally regarded as out of the scope of testing : they require different data and techniques.

Several approaches of testing are possible: in this work we consider dynamic testing, i.e. the execution of the program under test on a finite subset of its input domain and the interpretation of the obtained results. Other approaches are inspections or analysis of the text of the program; they are often called static techniques.

Obviously, testing methods strongly depend on the nature of the faults which are searched for. We consider here a large class of faults, namely discrepancies between the behaviour of a program and some functional specification. It means that non functional aspects such as performance or robustness are currently not considered in our framework.

There exist numerous criteria and strategies to guide the choice of the test data, i.e. the *selection* of a finite subset of the input domain called the test set. A test data criterion is a predicate *C(P, S, T)* which characterises those test sets *T* which are adequate for testing a program *P* with respect to a specification *S* according to the criterion. An example of a common (and weak) criterion is that *T* must ensure the execution of all the statements in *P*. When a criterion is only based on the text of the program, as in this example, the corresponding testing strategy is said to be structural. When the criterion is only based on the specification, the strategy is called functional. The program is considered as a “black-box” and the way it is written does not influence the test. Clearly, structural testing and black-box testing are complementary: structural testing cannot detect omissions; black-box testing cannot test every detail of the program. Currently, most research results on software testing are related to structural testing for historical reasons: without formal specifications it was difficult to study functional testing on theoretical grounds. It is this problem which is addressed in this paper.

Most test methods, and the corresponding selection criteria, consist in dividing the input domain of the program into subdomains and require the test set to include at least one element from each subdomain (these subdomains are not always disjoint, it is the case for the “all-statements” criterion mentioned above). Reasoning by cases on a formal specification seems a very natural way to define such subdomains. The approach presented here follows this idea. It is based on algebraic specifications and has been experimented on several significant case studies. Some underlying ideas have been reused for other kinds of formal specification or model: VDM in [13], finite state automata and labelled transition systems in [26], and Lustre programs [21].

The interpretation of the results of a test is often very difficult. This difficulty is known as the *oracle problem*: deciding whether the result of a test execution is acceptable or not requires a procedure (manual, automated, or both) which relies upon a knowledge of the expected results. For some problems, the expected results are not known for the whole input domain (for instance, it happens in numerical analysis); in some other cases the decision problem is undecidable (for instance the equivalence of the source and object programs when testing a compiler).

But even in less extreme situations, the problem is difficult as soon as the program yields the results in a way which may depends on some representation choices and makes the comparison with the specified results difficult. This is an important issue in the case of black-box testing, since the test is based on a specification which is (normally) more abstract than the program. Thus program results may appear in a form which is not obviously equivalent to the specificied results. This contradicts a common belief that the existence of a formal specification is sufficient to directly decide whether a test is a success: some more work is often needed. A solution to this problem is proposed in the case of algebraic specifications. It is based on a notion of observational equivalence of the specification and the program [28, 20].

The paper is organized as follows: part 2 presents a theory of black-box testing based on formal specifications and both the test data selection problem and the oracle problem are addressed in the case of algebraic specifications; part 3 reports briefly several applications: some case studies in algebraic specifications and the use of some aspects the approach for other formalisms.

2 A Theory of Testing

2.1 Specifications and Programs

Algebraic specifications are characterized as usual by a signature $\Sigma = (S, F)$, where S is a finite set of sorts and F a finite set of operation names with arity in S, and some axioms, i.e. a finite set Ax of Σ-formulas. We consider positive conditional axioms, i.e. the following form of formulas:

$$(v_1 = w_1 \wedge \ldots \wedge v_k = w_k) \Rightarrow v = w$$

with $k \geq 0$, where v_i , w_i , v and w are Σ-terms with S-variables.

A subset S_{obs} of observable sorts is distinguished among the sorts of S.

Let SP be such a specification and P be a program under test. Since we consider dynamic testing, we are interested by the properties of the computations by P of the operations mentioned in Σ; P must provide some procedure or function for executing these operations; the question is whether they satisfy the axioms of SP. Given a ground Σ-term t, we note t_P the result of its computation by P. We define now how to test P against a Σ-equation.

Definition 1: Given a Σ-equation ε, and a program P which provides an implementation for every operation name of Σ,

- a *test* for ε is any ground instantiation $t = t'$ of ε;
- a *test experiment* of P against $t = t'$ consists of the evaluation of t_P and t'_P and the comparison of the resulting values.

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 P, and we note it $P \models \Gamma$ where Γ is the test, i.e. a ground formula (deciding whether $P \models \Gamma$ is the oracle problem mentioned in the introduction; we postpone the discussion on the way it can be realised to section 2.3).

We can now introduce the definition of an exhaustive test of a program P against a specification SP.

Definition 2: Given a specification $SP = (\Sigma, Ax)$, the *exhaustive test set* for *SP*, noted $Exhaust_{SP}$ is the set of all well-sorted ground instances of all the Σ-axioms:

$$Exhaust_{SP} = \{ \Phi\sigma \mid \Phi \in Ax, \sigma = \{\sigma_s : var(\Phi)_s \to T_{\Sigma s} \mid s \in S\} \}$$

An exhaustive test of *P* against *SP* is the set of all the test experiments of *P* against the formulas of $Exhaust_{SP}$.

The definition of $Exhaust_{SP}$ is very close to (and is derived from) the notion of satisfaction of a set of Σ-equations by a Σ-algebra as it has been defined for a long time [17]. In particular, the fact that each axiom can be tested independently comes from this definition. But several points prevent the success of an exhaustive test of *P* against *SP* from being equivalent to the satisfaction of *SP* by *P*. This is true under some conditions on *P* which are discussed below.

It seems natural and convenient to consider P as defining a Σ-algebra, and we assume it for the moment. But this is far from being a weak assumption. It means that there is no influence of any internal state on the procedures which implement the operations of Σ: they behave as mathematical functions. As it has been pointed out in [22], it is possible to weaken this hypothesis on *P*. This point is related to the oracle problem and is discussed in section 2.3.

A second point is that in definition 2, the substitutions σ assign ground Σ-terms to the variables. It means that $Exhaust_{SP}$ is exhaustive with respect to the specification, not always with respect to the program. Thus, such a test ensures the satisfaction of *SP* by *P*only if all the values computable by *P* are reachable by T_Σ. This two points define a class of programs for which the result of an exhaustive test would be meaningful.

Definition 3: Given a signature Σ, a program *P* is *Σ-testable* if it defines a finitely generated Σ-algebra A_P. The Σ-testability of *P* is called the *minimal hypothesis* H_{min} of an exhaustive test of *P* against a specification of signature Σ.

The definitions of $Exhaust_{SP}$ and H_{min} provide a framework for developing a theory of black-box testing from algebraic specifications. Practical test criteria (i.e. those which correspond to a finite test set) will be described as stronger hypotheses on the program. Important properties such as *unbias* (correct programs are not rejected) and *validity* (only correct programs are accepted) can be characterized. This is done in the following subsections. Now, we list some remarks on these definitions where we discuss their adequacy and suggest some variants.

Remark 1: Strictly speaking, definition 1 above defines a *tester* rather than a test data: a test $t = t'$ is nothing else than the abstract definition of a program which evaluates *t* and *t'* via the relevant calls to the procedures of a program and compares the results; a test experiment of *P* is an execution of this tester linked to *P*. There is an interesting analogy with Brinksma's work on protocol testing [8]: there, from the specification of the process under test is derived a specification of a “canonical tester process”; the concurrent execution of the two processes performs the test experiment.

Remark 2: As said above, the definition of $Exhaust_{SP}$ comes from the notion of satisfaction of [17]. However, it does not correspond exactly to initial semantics of algebraic specifications since inequalities are not tested: it rather corresponds to loose semantics. It is possible to choose another definition. For instance, as suggested in [4] and applied by Dong and Frankl in the ASTOOT system [14], another possibility is to consider the algebraic specification as a rewriting system, following a “normal-form” 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. In [14] a bigger exhaustive test set is mentioned (but not used) which includes for every ground term the inequalities with other normal forms, following the definition of initial semantics.

Remark 3: An open question is how to deal with the case of test experiments which do not terminate (more precisely, the evaluation of t_P or t'_P does not terminate). Given the kind of specification that is considered here, where performance issues are not addressed, there is no way to make a decision in such a case… In practice, it is generally not a problem since more information is available on the expected behavior of the system.

Remark 4: The generalization of the theory presented here to partial operations is possible when the specification provides definition predicates which are completely specified, i.e they are defined for all the ground terms. In this case, it is possible to define the exhaustive test set as the restriction of $Exhaust_{SP}$ to those formulas which only contain equations with defined terms. This means that the exhaustive test checks the axioms for all the terms which must be defined. Other definitions of $Exhaust_{SP}$ are possible, depending on the considered semantics (for instance all partial algebras or only minimally defined ones [9]) and on some conventions on what is a correct behaviour of the program for undefined terms.

2.2 Selection and Hypotheses

$Exhaust_{SP}$ is obviously not usable in practice since it is generally infinite. One way to make it finite is to introduce stronger hypotheses on the behaviour of *P*. These *selection hypotheses* are the formal counterpart of some common test methods: for instance, the subdomain-based selection criteria mentioned in the introduction correspond to the determination of subdomains of the variables where the program is supposed to have the same behaviour. Assuming that, it is no more necessary to have all the ground instances of the variables but only one by subdomain. Such criteria are modelled in our framework by uniformity hypotheses.

Definition 4: Given a formula $\Phi(X)$ where X is a variable, a *uniformity hypothesis* on a subdomain D for a program P is the assumption:

$$(\forall t_0 \in D)\ (P \models \Phi(t_0) \Rightarrow (\forall t \in D)\ (P \models \Phi(t)\ )$$

The generalization to several variables is straightforward.

In the framework of black-box testing, the determination of such subdomains is guided by the specification, as we will see later. Other kinds of selection criterion correspond to other forms of hypothesis. For instance regularity hypotheses express the fact that it is sometimes enough to test a formula for terms under a certain “size”.

Definition 5: Given a formula $\Phi(X)$ where X is a variable, an a function of interest $|t|$ from ground terms into natural numbers, a *regularity hypothesis* for a program P is the assumption:

$$((\forall t \in T_{\Sigma})\ (\,|t| \leq k \Rightarrow P \models \Phi(t)\ ))\ \Rightarrow\ (\forall t \in T_{\Sigma})\ (P \models \Phi(t))$$

Some other kinds of useful hypothesis will be mentioned in section 3. The choice of the hypotheses is nothing else than the determination of a test strategy, of a test selection criteria. But the advantage of the notion of hypothesis is that it makes explicit the assumptions on the program which correspond to one strategy. A test set T should never be presented independently of its selection hypotheses: thus we use the notion of a *testing context* which is a pair (H, T) of a set of hypotheses and a set of tests. Now, we define some important properties which are required for testing contexts:

Definition 6: Given a specification $SP = (\Sigma, Ax)$, a testing context (H, T) is *valid* if, for all Σ-testable program P

$$H \Rightarrow (P \models T \Rightarrow P \models Exhaust_{SP})$$

Definition 7: Given a specification $SP = (\Sigma, Ax)$, a testing context (H, T) is *unbiased* if, for all Σ-testable program P,

$$H \Rightarrow (P \models Exhaust_{SP} \Rightarrow P \models T)$$

Assuming H, validity ensures that any incorrect program is rejected and unbias prevents the rejection of correct programs. By construction, the testing context $(H_{min}, Exhaust_{SP})$ is valid and unbiased. Another extreme valid and unbiased testing context is $(H_{min} \wedge P \models Exhaust_{SP}, \emptyset)$, which states that assuming that the program is correct, an empty test set is sufficient.

An interesting fact is that any context (H, T) where T is a subset of $Exhaust_{SP}$ unbiased. Conversely, if T contains a test which is not a consequence of SP, or which is in contradiction with it, any context (H, T) is biased.

It is clear that the interesting testing contexts are valid and unbiased and they are compromises between the two extreme examples given above. Intuitively, weak hypotheses correspond to large test sets and conversely. This naturally lead to the definition of a preorder on testing contexts [2]:

Definition 8: Let $TC1 = (H1, T1)$ and $TC2 = (H2, T2)$ be two testing contexts. $TC1 \leq TC2$ (pronounce "*TC2* refines *TC1*") iff:

(i) $H2 \Rightarrow H1$

(ii) $H2 \Rightarrow (P \models T2 \Rightarrow P \models T1)$

$\leq$ means that in the refined testing context the hypotheses on the program are stengthened, and, assuming these refined hypotheses, the refined test set reveals as many faults as the original one. $\leq$ is reflexive, transitive, but not anti-symmetric. It makes it possible to build valid contexts since it preserves validity:

Proposition: If $(H_{min}, Exhaust_{SP}) \leq (H, T)$, then (H,T) is valid.

The proof results directly from the definitions. This proposition and the above remarks on unbiased contexts provide the bases of a method to obtain valid and unbiased testing contexts: the starting point is $(H_{min}, Exhaust_{SP})$ and by successive refinements, new hypotheses are added, and the test set is restricted to a subset of the previous one (thus of $Exhaust_{SP}$); for every step the property (ii) of definition 8 must be verified.

The choice of the selection hypotheses can be guided by the specification. For instance, given an axiom of the form: $(v_1 = w_1 \wedge \ldots \wedge v_k = w_k) \Rightarrow f(\tau(X)) = w$, where X is a list of variables also occuring in the v_i, w_i and w, the domain defined by the precondition can be considered as a uniformity subdomain. This gives one test for the axiom, and the instanciations of the variables are any solution of the set of equations of the precondition. If this coverage of the axiom is considered not sufficient, weaker uniformity hypotheses on subdomains can be obtained by *unfolding* the axiom.

We just give an example of one unfolding step and send the reader to [3] for a formal presentation. Assume that v_1 is of the form $g(\mu(X))$, that there is no other occurence of g in the axiom, and that there is the following axiom somewhere in the specification : $cond(Y) \Rightarrow g(\nu(Y)) = w'$; then it is possible to compose the preconditions of the two axioms, and it will result in a more complicated axiom, namely : $cond(Y) \wedge \mu(X) = \nu(Y) \wedge w' = w_1 \wedge \ldots \wedge v_k = w_n \Rightarrow f(\tau(X)) = w$. Note that this axiom is never explicitely constructed : the specification remains unchanged. Unfolding is just used to build smaller uniformity subdomains by composition of the preconditions of the axioms. There will be as many tests of the original axiom as possible unfoldings and a set of weaker uniformity hypotheses. Now the problem of choosing uniformity subdomains is transformed into the problem of deciding when to stop unfolding, since it is well-known that unfolding recursive definitions of functions on recursive domains results in the enumeration of the domain. One possible way is to introduce some regularity hypothesis or some other uniformity hypotheses.

This kind of symbolic manipulation of the specification is implemented by a tool, the LOFT system [23]. The kernel of this system is an equational resolution procedure since the implementation of uniformity hypotheses requires to solve conjunctions of equations in the theory defined by the specification.

2.3 The Oracle Problem

In definition 1, we have stated that a test experiment of P against $t = t'$ consists of the evaluation of t_P and t'_P and the comparison of the resulting values. This comparison is obvious to perform when the sort s of t and t' corresponds exactly to a type of the programming language. Under the (weak) hypothesis that this comparison is correctly implemented by the compiler, we can use the equality of the programming language, noted $eq_{s,P}$ as a valid and unbiased oracle, i.e. test whether $P \models t = t'$ by computing $eq_{s,P}(t_P , t'_P)$.

We call such an hypothesis an *oracle hypothesis*. Some sorts for which it is sound to assume that their equality is correctly implemented are the predefined sorts of the programming languages (booleans, integers, ...). We distinguish these sorts as *observable*: as in [20] they will be used to observe the implementation of the other sorts.

It is more problematic to have such an hypothesis on data types defined by the programmer to represent more elaborated sorts of the specification. Even if an equality function is available for these sorts, it is a part of the program under test and it is possible for this function or the representation to be erroneous. The naive idea of testing the equality of all the observable components of the representation is wrong: the folk literature on abstract data types provide numerous examples where different representations correspond to the same abstraction (see [3] for a nasty variant of the good old stack example).

One solution is to refer to an observational equivalence for defining such equalities: an observation is a computation which has a non observable value as argument and returns an observable value. Clearly, if there is a way, using P, to get different results from the same observation of two non observable values, these values are different. Conversely, if there is no observable difference, they can be considered as equal.

Definition 9: Given a signature Σ, a *Σ-context* is a Σ-term which contains *exactly one* variable. If the variable is of s sort, we say it is a *Σ-context over s*. We call an *observable context* a context of observable sort where the variable is of non observable sort and we note $OC_{\Sigma}[s]$ the set of observable contexts over s for a signature Σ.

Given the non observable test $t = t'$ where t and t' are of sort s, it can be transformed into an (often infinite) set of observable tests:

$$\{ C(t) = C(t') \mid C \in OC_{\Sigma}[s]\}$$

It is possible to only consider minimal observable contexts, i.e. those which do not contain any observable context, assuming that the implementation of the operations of the observable sorts preserves equality. It is a slightly stronger oracle hypothesis than the correctness of the equality implementation, but even stronger ones, such as the correctness of the implementation of observable sorts, are very likely to be sound if, as suggested above, the observable sorts correspond to the predefined types of the programming languages.

Anyway, the set of observable tests is very likely to be infinite. A natural way to get finite observable test sets is to follow a similar approach as in the previous section: we may introduce some hypotheses to reduce the infinite set of contexts to a finite one. However, an erroneous decision in favor of equality has different consequences depending on the position of the considered equation in the axiom:

- if the equation is the *conclusion* of a positive conditional axiom, if the oracle hypotheses are not satisfied by the program, the finite set of observable tests may be successful for an incorrect program;
- if the equation occurs in the *precondition* of an axiom, and if its test is erroneouly decided successful, the conclusion of the axiom is required to be valid despite the fact that it is not required by the specification, and there is a risk to reject correct programs.

The first case is acceptable in our framework since their is no problem of bias, and the validity of the testing context is preserved (remind that validity is defined by: *assuming the hypotheses*, the success of the test set implies the satisfaction of the specification). But the second one is not acceptable since it corresponds to a biased testing context. This leads to a restriction on the kind of considered algebraic specifications: it is possible to use observable contexts as oracle only for those positive conditional specifications where all the equations in preconditions are of observable sorts. Under this condition, we define an observable exhaustive test set for a specification:

Definition 10: Given a positive conditional specification $SP = (\Sigma, Ax)$, where is distinguished a subset S_O of observable sorts, where all the equations occuring in the preconditions of the axioms of Ax are of observable sort, the *observable exhaustive test set* for SP, noted Obs_{SP} is:

$$\begin{gathered} Obs_{SP} = \{ L \Rightarrow C(t) = C(u) \mid L \Rightarrow t = u \in Exhaust_{SP}, \\ t \text{ and } u \text{ of sort } s \notin S_O,\ C \in OC_{\Sigma}[s] \} \cup \\ \{ L \Rightarrow t = u \mid L \Rightarrow t = u \in Exhaust_{SP},\ t \text{ and } u \text{ of sort } s \in S_O \} \end{gathered}$$

An *observable exhaustive test* of *P* against *SP* is the set of all the test experiments of *P* against the formulas of Obs_{SP} .

The notion of ∑-testability (definition 3) must be revised accordingly in order to ensure that the success of an observable exhaustive test set is meaningful: the conditions on the observable sorts must be added; H_{min} can be weakenened (the definition given in [3] turned out to be too strong as pointed out in [22]). It must be noted that stating definition 10, we have changed the notion of satisfaction of a specification *SP* by a program *P* into a notion of behavioural satisfaction (more precisely, the “behavioural satisfaction I” of [25]), and we have kept the fact that the exhaustive test only considers those values which are finitely generated by *∑*. Thus there is no more need that *P* defines a finitely generated ∑-algebra: it is sufficient that it can be observed as a finitely generated ∑-algebra via the observable contexts.

Definition 11: Given a signature *∑* where is distinguished a subset S_o of observable sorts, a program *P* is *O∑-testable* if:

- for every observable sort *so*, *P* provides a correct implementation of equality;
- the behaviour of *P* is observationally equivalent via the observable contexts to a finitely generated ∑-algebra

The O∑-testability of *P* is called the *minimal hypothesis* $OBSH_{min}$ of an observable exhaustive test of *P*.

We are now in the position of defining validity and unbias of testing contexts in this new framework:

Definition 12: Given a positive conditional specification $SP = (\Sigma, Ax)$, where is distinguished a subset S_o of observable sorts, where all the equations occuring in the preconditions of the axioms of *Ax* are of observable sort, a testing context *(H, T)* is *observationally valid* if, for all O∑-testable program *P*

$$H \Rightarrow (P \models T \Rightarrow P \models Obs_{SP})$$

Definition 13: Given a specification $SP = (\Sigma, Ax)$, a testing context *(H, T)* is *observationnaly unbiased* if, for all O∑-testable program *P*,

$$H \Rightarrow (P \models Obs_{SP} \Rightarrow P \models T)$$

Similarly to the previous subsection, the testing context ($OBSH_{min}$, Obs_{SP}) can be used as a starting point to build observationally valid and unbiased testing contexts by addition of new hypotheses and restriction of the test set in a way satisfying requirement (ii) of definition 8: any testing context *(H,T)* where *T* is a subset of Obs_{SP} is observationally unbiased, and the ≤ preorder preserves observational validity.

Remark 5: It has been suggested to choose the observable sorts among the predefined types of the programming language. It is clear that other possibilities can arise: it may be possible to correctly decide whether two values of a type are equal even is there is not a built-in equality operation in the programming language.

Remark 6: It is interesting to note that the restriction on the form of the specification introduced by definition 10 is mentioned in [25] as necessary to have a sensible definition of behavioural satisfaction based on the concept of observable consequence (our $C(t) = C(t')$ are observable consequences of *SP*). It is clear that, depending on the kind of considered program, other observable exhaustive test sets could be defined, with some other kind of observation and thus a different notion of behavioural satisfaction. In such cases, there is no reason to have the same restriction.

Remark 7: Another reasonable requirement on the specification is that it should be hierarchically consistent with respect to the specification of the observable sorts. However, in our theory, it is quite possible to test a program against an inconsistent specification, but the observable exhaustive test fails as soon as the program satisfies $OBSH_{min}$.

Remark 8: Practically, Σ-testability requires that the program provides an implementation of every operation of Σ, which must behaves as a function; moreover the values computable by the program must be finitely generated by Σ. One way to ensure functional behaviours of the operations is to forbid the use of global variables and any side effects in the program. One way to ensure the last point is to require, as in [3], that the program exports exactly the operations of Σ. These conditions are sufficient but too strong to be realistic and it is important to look at more flexible ones. OΣ-testability slightly relaxes the constraint of a functional behaviour of the operations into an "observational" functional behaviour. Besides, it is clearly possible to allow the program to define the operations of Σ up to a renaming; it is also possible to allow the program to export more operations, or less operations (if some hidden operations are used in the specification). A first discussion of the conditions under which it is possible can be found in [22].

Remark 9: These minimum hypotheses are static properties of the program. Some of them are (or could be) checkable: currently the tools for static checking of programs are often independant of the specifications. However, [16] reports some promising experiments on the use of specifications for statically detecting violations of abstraction barriers.

2.4 Equational test sets

When the specification is sufficiently complete with respect to the observable sorts, it is possible to consider another exhaustive test set where all the elementary tests are equations [2].

Definition 14: Given a positive conditional specification $SP = (\Sigma, Ax)$, where is distinguished a subset S_o of observable sorts, where all the equations occuring in the preconditions of the axioms of Ax are of observable sort, the *exhaustive observable equational test set* for SP, noted $EqObs_{SP}$ is:

$$EqObs_{SP} = \{ t = u \mid t_1 = u_1 \wedge \ldots \wedge t_k = u_k \Rightarrow t = u \in Obs_{SP}, \\ \forall i = 1, \ldots, k,\ Ax \vdash_{EQ} t_i = u_i \}$$

where $\vdash_{EQ}$ corresponds to the usual equational calculus.

The testing context $(OBSH_{min}, EqObs_{SP})$ is observationally unbiased and valid if SP is sufficiently complete. Thus it can be use as a starting point for the selection of finite test sets by the addition of selection hypotheses as shown in section 2.2. It is what is done by the LOFT system.

3 Applications

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 second one reports some interesting attempts to transport some aspects of the theory to other formal approaches, namely VDM, automata, and Lustre.

3.1 Some Case Studies with Algebraic Specifications

A first experiment, performed at LRI by Pierre Dauchy and Bruno Marre, was on the on-board part of the driving system of an automatic subway. An algebraic specification was written [12] . Then two modules of the specification were used for experiments with LOFT. The experiment is reported in details in [11]. We just give the conclusions here. It must be noted that this work was performed with the certification agency (INRETS), not with the development team of the system.

The first module was the control of the door opening ; there were 25 axioms, some of them with rather long and complex preconditions. The choice of the uniformity subdomains was done in a standard way : for instance unfolding ≤ into = and < and stopping, unfolding false conjunctions and true disjunctions into three cases, but forbidding any decomposition leading to meaningless enumerations. The total number of tests turned out to be reasonable : 254. The good surprise was that 230 of these tests were related to the emergency stop, and that some of them presented some conjunctions of cases which were not yet considered in the certification process.

The second module was the overspeed control. There were 6 axioms in the main module which used 4 other modules. A first experiment was performed with the same choices as for the previous module, and reasonable test sets were selected for most of the axioms (between 1 to 40); but some problems arose when trying to treat the following axiom :

limspeed(S) = min4 (target-speed(S), stop-speed(S),
imposed-lim-speed(S), way-speed(S))

where *S* is the state of the train and *min4* is the minimum of its four arguments. A brute application of the same strategy as above to the four arguments of *min4* would lead to 25920 tests… The point is that a lot of useless decompositions are performed, for instance if *target-speed(S)* is the minimum, it is useless to test all the possible orders on the three other speeds. Thus another testing strategy was tried: first perform the decomposition of the *target-speed* operation, then add to the obtained subdomains the three equations describing the comparisons with the other speeds and return one value in this domain. This example demonstrates that the choice of a testing context is an interactive process: the LOFT system aims at assisting this process, by guiding the choice of the uniformity subdomains and yielding unbiased and valid testing contexts

A second experiment is reported in [24] and was performed within a collaboration between LRI and the LAAS laboratory in Toulouse. Our colleagues from LAAS have a good experience in evaluating the quality of test sets by mutation. The experiment was performed on a rather small piece of software written in C, which was extracted from a nuclear safety shutdown system. It was the filtering procedure which aims at checking the validity of successive measures in order to eliminate doubtful ones. For some previous work, 1345 mutants of this procedure have been built by the LAAS team using classical fault injection methods.

The algebraic specification was written in a data-flow like style and contained 30 axioms. 5 different test sets of 282 tests were selected using the LOFT system, with the same hypotheses : we were interested in studying both the quality and the stability of the method with respect to the arbitrary choices in the uniformity subdomains. The score, i.e. the rate of rejected mutants, turned out to be rather stable, from 0.9651 to 0.9784, and good for a black-box strategy. These results were better than the ones of the all-paths structural strategy, which is supposed to be the most powerful structural testing strategy.

The Software Engineering Laboratory at EPFL in Lausanne has developped and is continually extending a library of components in Ada. The components are used for teaching and in industry; most of them have been released as publicly available software. A characteristic of this library is the existence of numerous variants (with respect to parameter passing, storage properties, etc) of some components. Such a set of variants is called a family. An experiment of “intensive” testing of a family have been led by Didier Buchs and Stéphane Barbey [1]. 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 [23].

The ASTOOT approach has been developped by Phyllis Frankl and her team at the Polytechnic University in New-York [14]. The addressed problem is the test of object-oriented programs: classes are tested against algebraic specifications. A set of tools has been developped; we focus here on the test cases generation issues. As mentioned in remark 2, a different choice has been made for the exhaustive test set, which is the set of equalities of every closed term with its normal form. Thus, 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 is solved by introducing a notion of observational equivalence between objects of user-defined classes, which is based on minimal observational contexts, and by approximating it. The test case selection is guided by an analysis of the conditions occuring in the axioms; the result is a set of constraints which must be solved manually. It is encouraging to note that this project was led independently from our research, but that it turned out, a posteriori, that the theory presented here nicely fits to describe it, even when different basic choices were made.

3.2 Other Formalisms

In section 2 we followed several times the same procedure to build slightly different formal bases for program testing against an algebraic specification: we stated a notion of test, a notion of exhaustive test and some minimal hypotheses to ensure validity and unbias. A reasonable conjecture is that this procedure is applicable to other kinds of formal specification. In the case of VDM, a first interesting step has been performed by Jeremy Dick and Alain Faivre within a project, led at Bull Corporate

Research Centre, on automating the generation of test cases from VDM-SL specifications [13].

The formulae of the specification 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 the equivalent of the axioms unfolding presented in section 2.2. Uniformity an 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 subrelations. 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.

A tool has been developed to assist this process. However, since the kind of considered formula is more general than in our case, it is not possible, in general, to solve the returned constraints: the tool is more a test cases generator than a test data generator.

This work makes numerous references to some of the important notions and techniques presented in section 2: uniformity and regularity hypotheses, unfolding. What is currently missing is a notion of exhaustive test set, and its connex concepts, testability, validity and unbias. A very tentative idea is to consider that a specification defines an infinite automaton, and to take the test suites which exercise all the paths of this automaton as an exhaustive test set. Clearly, it deserves more investigation and more thought about the kind of semantics to be considered.

In a thesis prepared at the France-Telecom CNET laboratory in Lannion [26], Marc Phalippou has studied conformance testing of telecommunication systems with respect to a formal model. His work is based on IOSM, as Input-Output State Machines, which are a variant of labelled transition systems. He has systematically studied the kinds of selection hypothesis which are used, or could be useful in this framework. First he shows that most existing methods for generating test data from automata are based on regularity hypotheses (more exactly, on the counterpart, in the automata framework, of the definition presented here: its form is rather different). Then he shows how uniformity hypotheses can be seen as congruence on automata, and he introduces two variants, weak and strong uniformities. He suggests several other kinds of hypothesis which provide a formal expression of some interesting testing strategies. One of them, the fairness hypothesis is of quite general interest since it corresponds to a notion of testability for non deterministic systems.

In another thesis, prepared jointly at CEA and LRI, the use the LOFT system to assist the test of Lustre programs has been investigated [21]. Lustre is a description language for reactive systems which is based on the synchronous approach [10]. An algebraic semantics of Lustre has been stated and entered as a specification in LOFT. A Lustre program is considered as an enrichment of this specification, just a specific axiom to be tested. It is too early to decide of the practical interest of this work, but this way of adapting new formalisms to the system is interesting.

Acknowledgements

The content of this article is directly indebted to joint work and many stimulating discussions with Gilles Bernot, Bruno Marre and Pascale Le Gall. Bruno Marre has been very helpful in the preparation of this paper. The discussions with Phyllis Frankl during her visit to Orsay in June 94 influenced this presentation in a significant way.

References

1. Barbey S., Buchs D., Testing Ada abstract data types using formal specifications, in Ada in Europe, proc. 1st Int. Eurospece-Ada-Europe Symposium, Copenhagen, Sept. 1994, LNCS n°887, Springer-Verlag, 1994, pp. 76-89.
2. Bernot G., Testing against formal specifications: a theoretical view, TAPSOFT'91 CCPSD proceedings, LNCS n° 494, Springer-Verlag, Brighton, 1991, pp. 99-119.
3. Bernot G., Gaudel M-C., Marre B., Software testing based on formal specifications : a theory and a tool, Software Engineering Journal, vol. 6, n° 6, Nov. 1991.
4. Bernot G., Gaudel M-C, Marre B., A Formal Approach to Software Testing, 2nd International Conference on Algebraic Methodology and Software Technology (AMAST), Iowa City, Mai 1991,Worshops in Computing Series, Springer-Verlag, 1992.
5. Bougé L., A contribution to the theory of program testing, Theoretical Computer Science, vol. 37, 1985, pp. 151-181.
6. Bougé L., Choquet N., Fribourg L., Gaudel M.-C., Application of PROLOG to test sets generation from algebraic specifications, TAPSOFT'85 proceedings, LNCS n°186, Springer-Verlag, Berlin, 1985, pp. 246-260.
7. Bougé L., Choquet N., Fribourg L., Gaudel M.-C., Test set generation from algebraic specifications using logic programming, Journal of Systems and Software, vol.6, n°4, pp. 343-360, 1986.
8. Brinksma E., A theory for the derivation of tests, 8th International Conference on Protocol Specification, Testing and Verification, Atlantic City, North-Holland, 1988.
9. Broy M., Wirsing M., Partial Abstract Types. Acta Informatica, 3, 1982, pp. 47-64.
10. Caspi P., Halbwachs N., Pilaud D., Plaice J., Lustre: a declarative language for programming synchronous systems, 14th ACM symposium on Principle of Programming Languages, Munich, 1987, pp. 178-188.
11. Dauchy P., Gaudel M-C, Marre B., Using Algebraic Specifications in Software Testing : a case study on the software of an automatic subway, Journal of Systems and Software, vol. 21, n° 3, June 1993, pp. 229-244.
12. Dauchy P., Ozello P., Experiments with Formal Specifications on MAGGALY, Second International Conference on Applications of Advanced Technologies in Transportation Engineering, Mineapolis, Aug. 1991.
13. Dick J., Faivre A., Automating the generation and sequencing of test cases from model-based specifications, FME'93, LNCS n°670, Springer-Verlag, 1993, pp. 268-284.
14. Dong R. K., Frankl Ph. G., The ASTOOT approach to testing object-oriented programs, ACM Transactions on Software Engineering and Methodology, vol. 3, n° 2, Apr. 1994.
15. Dssouli R., Bochmann G., Conformance testing with multiple observers, in Protocol Specification Testing and Verification VI, North-Holland 1987, pp. 217-229.

16. Evans D., Using specifications to check source code, Master thesis, MIT Laboratory for Computer Science, 1994.

17. Goguen, J.A., Thatcher, J.W. and Wagner E.G., An initial algebra approach to the specification, correctness and implementation of abstract data types, in Current Trends in Programming Methodology, Vol.4: Data Structuring, edited by R.T. Yeh, pp. 80-149, Prentice-Hall, 1978.

18. Goodenough J. B., Gerhart S., Towards a theory of test data selection, IEEE Transactions on Software Engineering, vol. SE-1, n° 2, June 1975.

19. Gourlay J., A mathematical framework for the investigation of testing, IEEE Transactions on Software Engineering, vol. SE-9, n° 6, pp. 686-709, Nov. 1983.

20. Hennicker R., Observational implementations of algebraic specifications, Acta Informatica, vol. 28, n° 3, pp. 187-230, 1991.

21. Hsiao N. C., Sélection de test de propriétés de sûreté à partir d'une modélisation algébrique de programme Lustre, Thèse de l'Université de Paris-Sud, Orsay, 1994.

22. Le Gall P., 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, Thèse de l'université de Paris-Sud, LRI, Orsay, 1993.

23. Marre B., LOFT, a tool for assisting test data selection from algebraic specifications, in these proceedings.

24. Marre B., Thévenod-Fosse P., Waeselink H., Le Gall P., Crouzet Y., An experimental evaluation of formal testing and statistical testing, SAFECOMP'92, Zürich, Oct. 1992.

25. Orejas F., Implementation and behavioural equivalence: a survey, 8th WADT/3rd COMPASS Workshop, Doudan, 1991, LNCS n° 655, Springer-Verlag, 1993, pp. 93-125.

26. Phalippou M., Relations d'implantation et hypothèses de test sur des automates à entrées et sorties, Thèse de l'université de Bordeaux 1, Sept. 1994.

27. Pitt D.H., Freestone D., " The derivation of conformance tests from LOTOS specifications", IEEE Transactions on Software Engineering, vol. 16, n°12, Dec. 1990, pp. 1337-1343.

28. Sannella D. T., Tarlecki A., On observational equivalence of algebraic specifications, J.C.S.S., vol.34, pp. 150-178, 1987.