A Formal Approach to Software Testing

Gilles Bernot[1], Marie-Claude Gaudel[2] and Bruno Marre[3]

[1] LIENS, URA CNRS 1327, Ecole Normale Supérieure, 45 rue d'Ulm, F-75230 Paris cedex 05, email: bernot@frulm63
[2] LRI, URA CNRS 410, Université PARIS-SUD, F-91405 Orsay cedex, email: mcg@frlri61.bitnet
[3] LRI, URA CNRS 410, Université PARIS-SUD, F-91405 Orsay cedex, email: marre@frlri61.bitnet

Abstract. This paper addresses the problem of the formalization of software testing. More precisely, we are concerned with the use of formal specifications for guiding the testing activity. This kind of software testing is often called "functional testing." Formal specifications provide a basis for what we call "formal testing": i.e. the definition of testing in a formal framework which makes explicit the relationship between testing and correctness. Besides, it provides some grounds for the study of the complementarity of testing and proving.

Introduction

Software testing is sometimes considered as intrinsically empirical, without possibility of theoretical grounds. There is some evidence that it is not the case: see for instance [1] and [2] for some accounts of the research in the area and [3] or [4] for some theoretical foundations.

This paper discusses the use of formal specifications for guiding the testing activity and for formalizing it. It is well-known that one of the advantages of formal specifications is to give the possibility of correctness proofs of programs. We believe that formal specifications also provide a basis for what we call "formal testing": i.e. the definition of testing in a formal framework which makes explicit the relationship between testing and correctness. Besides, it provides some grounds for the study of the complementarity of testing and proving.

Moreover, such a formal approach to testing has proved to be a good basis for automatizing the selection of test data sets from formal specifications [5], and provides a better understanding of some common test practices.

We present briefly here some new developments of a research activity which has been reported in [6, 7, 8, 9, 5, 10], and more recently in [11] and [12]. Our previous results were specialized to positive conditional algebraic specifications. We show in this paper that it is possible to work in the more general framework of institutions [13]. It makes it possible to consider a larger class of formal specifications; it makes clearer the relation between the semantics of the formal specifications and the corresponding testing process; moreover, changing of institution often correspond to changes in the kind of properties expressible in the formal specification: thus it allows to test the same program against different points of view.

1 Formal specifications and correctness

Briefly, a formal specification method (institution) is given by a *syntax* and a *semantics*.

- The syntax is defined by a notion of *signature*; with each signature Σ is associated a set of *sentences* Φ_Σ. Φ_Σ contains all the well-formed formulas built on Σ, some variables, some logical connectives and quantifiers.
- The semantics is defined by a class of *Σ-interpretations*, Int_Σ, and a *satisfaction relation* on $Int_\Sigma \times \Phi_\Sigma$ denoted by $\models$. For each Σ-interpretation A and for each Σ-formula ϕ, "$A \models \phi$" should be read as "A satisfies ϕ".

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

The class of interpretations *satisfying* SP is called the class of *models* of SP and is denoted by $Mod(SP)$:

$$Mod(SP) = \{ A \in Int_\Sigma \mid A \models Ax \}$$

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

Let SP be a formal specification and P be a program. It is possible to verify (by proving or by testing) the adequacy or inadequacy of P with respect to SP if the semantics of P and SP are expressible in some common framework. As we are interested in dynamic testing[4], this role is ensured here by the concept of an interpretation for a given signature: intuitively a Σ-interpretation is a set of values plus, for each name in the signature Σ, an operation of the relevant arity on these values. We consider that P defines a Σ-interpretation M_P. Then, the question of the correctness of P with respect to SP becomes: does M_P belong to the class of models of SP ?

2 Testing a program against a formula

In our framework, a *test data*[5] is a Σ-formula $\phi(X)$. As said above, $\phi(X)$ is a well-formed composition of some predicates, logical connectives, operation names of Σ, and variables in X. A test data is *executable* by a program P if it is a ground formula and if P actually defines a Σ-interpretation, i.e. P provides an implementation of every operation of the signature. Under these conditions, *running a test* ϕ consists of computing by M_P the operations of Σ which occur in ϕ and checking that the results returned by M_P satisfy the property required by

[4] i.e. test which consists of several executions of the program on a finite subset of its input domain, while static testing relies on some analysis of the program text.

[5] In protocol testing, what we call "test data" is commonly called "test purpose".

the connectives. For instance, let *f, g, h, a, b* belong to Σ, let $\vee$ be a connective, *x,y* some variables; and $\phi(x, y)$ the following formula:

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

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

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

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

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

There are some cases where it is not possible to decide whether or not an execution returns a correct result, mainly for reasons of insufficient observability of M_P: a property required by the specification may not be directly observable using the program under test (see for instance [14]). It is an aspect of the so-called *oracle problem*: the oracle is some decision process which should be able to decide, for each test data ϕ whether ϕ is successfully run or not when submitted to the program P. Providing such an oracle is not always trivial ([15, 16]) and may be impossible for some test data. An example, among many others, is how to decide that the result of a compiler for a given source program is correct, i.e. that the generated code is equivalent, in some way to be defined, to this source program.

3 Exhaustive data sets, Hypotheses

Let us come back to the notion of correction and the satisfaction relation. As an example of an institution, let us consider equational algebraic specifications as defined in [17].

> **Example 1:** In [17] the satisfaction relation is stated as
> " A Σ-equation is a pair $e = < L, R >$ where $L, R \in \mathcal{T}_\Sigma(\mathcal{X})_s$ for some sort s. Let $var(e) = var(L) \cup var(R)$ ... A Σ-algebra A *satisfies* e iff $\overline{\theta}(L) = \overline{\theta}(R)$ for *all* assignements $\theta : Y \to A$ where $Y = var(e)$. If E is a set of Σ-equations, then A *satisfies* E iff A satisfies every $e \in E$ ".
> (Where $\overline{\theta}$ is the unique extension of θ to $\mathcal{T}_\Sigma(\mathcal{X})$).

For a given specification, this definition naturally leads to the following test data set:

Definition:
given an equational algebraic specification $SP = < \Sigma, E >$, the *exhaustive* test data set, $Exhaust_{SP}$, is the set of all ground instances of all the equations in E.

$$Exhaust_{SP} = \{\sigma(\phi) \mid \phi \in E \ , \ range(\sigma) = \mathcal{T}_\Sigma\}$$

where $\mathcal{T}_\Sigma$ is the set of ground terms on Σ.

This test set is exhaustive with respect to the specification, not with respect to the program, since we limit the assignments to those values of M_P which are denotable by some term of $\mathcal{T}_\Sigma$. It means that the success of this test set will ensure correctness only if M_P is finitely generated with respect to Σ. Thus we have introduced an additional *hypothesis* on M_P (the first one was that M_P is a Σ-interpretation). We note this hypothesis $Adequate_\Sigma(M_P)$[6]. Let us assume that we have a correct oracle procedure named *success*; in this institution, as "=" is the only predicate, the oracle is a procedure which is able to decide, given two values of M_P , whether or not they represent the same abstraction in SP (cf. the *identify* step in [18]). Now, we have, *assuming* $Adequate_\Sigma(M_P)$:

$$success(Exhaust_{SP}) \iff M_P \in Mod(SP)$$

$Exhaust_{SP}$ is obviously not usable in practice since it is generally infinite. One way to make it practicable, i.e. finite, is to make stronger hypotheses on the behaviour of M_P. These *testing hypotheses* represent and formalize common test practices; for instance, identifying subdomains of the variables where the program is likely to have the same behaviour; in this case, it is no more necessary, assuming the hypothesis, to have all the ground instances of the variables, but only one by subdomain. As an example, when testing a stack implementation, it is commonly assumed that its behaviour, when pushing a value on a given stack, then poping, does not depend on the value. Such hypotheses are called *uniformity hypotheses*. There are other possible patterns of hypotheses, for instance *regularity hypotheses*, etc [12].

The choice of these hypotheses is driven by the specification. For instance, it is shown in [10] and [19] how unfolding techniques can be used to identify uniformity subdomains. Besides, this choice also depends on the acceptable size of the test set: as usual in testing, the problem is to find a sound trade-off between cost and quality considerations.

4 Validity, Unbias and Practicability

The hypotheses we have mentioned have the property that, starting from the axioms of the specification, they allow one to select executable test sets which are *valid* and *unbiased*.

[6] It is not surprising to have such assumptions in a specification-based testing approach. They just express that the program under test, which is seen as a "black box" should not be too far from the specification; for instance, they have the same signature; the program modules do not export additional operations.

A test set T is *valid* if, assuming some hypotheses H :

$$M_P \models T \implies M_P \in Mod(SP)$$

T is *unbiased* if, assuming H :

$$M_P \in Mod(SP) \implies M_P \models T$$

Similarly, an oracle *success* is *valid* if, assuming H, for every formula ϕ either $success(\phi)$ is undefined, or:

$$success(\phi) \implies M_P \models \phi$$

success is *unbiased* if, assuming H, for every formula ϕ either $success(\phi)$ is undefined, or:

$$M_P \models \phi \implies success(\phi)$$

Validity expresses that assuming the hypotheses, uncorrect programs are detected; unbias is the converse property, i.e. correct programs are not rejected.

A sufficient condition to ensure validity and unbias is to select test sets which are subsets of the exhaustive test set, *derived from the hypotheses*. However, this limitation is sometimes too strong. For instance, the need of an oracle may lead to build an observable exhaustive test data set which is not a subset of the exhaustive data set (it is however, valid and unbiased); or it is sometimes efficient to perform some simplifications of the formulas of the data set (see [12] sections 2.3 and 2.4).

The theoretical framework presented here introduces the important idea that a test data set cannot be considered (or evaluated, or accepted, etc) independently of some hypotheses and of an oracle. Thus, we define a *testing context* as a triple (*H,T,success*) where T is the test data set, H is a set of hypotheses on M_P and *success* is an oracle, i.e. a predicate partially defined on Φ_Σ (see [11] and [12] for more details).

The aim of test data selection is then to build *practicable* testing contexts. A testing context (*H,T,success*) is practicable if: T is finite; *success* is defined on all the test data in T; and assuming the hypotheses H, T and *success* are both valid and unbiased.

5 The Refinement Preorder

A starting point of the "testing elaboration process" directly results from the specification; it is called the *basic testing context*: (*Adequate*$_\Sigma$, *the axioms*, *the undefined oracle*). Then, the *refinement preorder* allows to derive a practicable testing context from the basic testing context (by successive refinements).

The refinement preorder:
Let $TC_1 = (H_1, T_1, success_1)$ and $TC_2 = (H_2, T_2, success_2)$ be two testing contexts. TC_2 *refines* TC_1 means that:

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

$$H_2 \implies H_1$$

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

$$\textit{Under the hypotheses } H_2,\ M_P \models T_2 \implies M_P \models T_1$$

– "under the hypotheses H_2, the oracle $success_2$ is more defined than $success_1$ and reveal as many errors as $success_1$ does"
Under the hypotheses H_2, if $success_1(\phi)$ is defined then $success_1(\phi)$ is defined too and

$$success_2(\phi) \implies success_1(\phi)$$

Example:
Let us assume that $TC_1 = (H_1,\ T \cup \{\varphi(x,s)\},\ success)$ where $\varphi(x,s)$ is the formula: $pop(push(x,s)) = s$, the variables x and s being respectively of types *natural numbers* and *stacks*. Let $Unif_x(\varphi)$ be the uniformity hypothesis mentionned at the end of section 3. This hypothesis allows us to replace the occurrence of x in $\varphi(x,s)$ by one arbitrary chosen value, e.g. 3. Thus $TC_2 = (H_1 \wedge Unif_x(\varphi),\ T \cup \{\varphi(3,s)\},\ success)$ is a refinement of TC_1, whatever H_1 and *success* are. Some other examples of refinements are given in [12], including oracle refinements.

It is not difficult to prove that if $(H, T, success)$ refines the basic testing context then T is valid. Moreover if T is a subset of one of the exhaustive test sets mentioned above and in Section 6, then it is unbiased. Validity and unbias of the oracle *success* is somewhat more difficult to ensure in general [11][12], however these oracle properties will be trivially ensured when considering the institution of "observational semantics of algebraic specifications" described in Section 6. Consequently, our refinement preorder provides a way for deriving *practicable* testing contexts from the basic testing context: it is sufficient to stop the refinement process when T is finite and included in the definition domain of the oracle *success*. Indeed, in [10] and [12] we present a tool which follows this refinement preorder to automatically produce test data sets from algebraic specifications.

6 Changing of Institution

We have shown on Example 1 how to derive, in the "ADJ institution" a triple (*minimal hypothesis, exhaustive test set, equality oracle*) in a canonical way. In the sequel we will call this triple $(Hmin_{ADJ}, Exhaust_{ADJ,SP}, Eq_{ADJ})$. It is interesting to look at such derivations for some other semantic approaches of algebraic specifications.

Example 2 (operational semantics of algebraic specifications):
Let us consider that the semantics of an algebraic specification $SP = < \Sigma, E >$ is the term rewriting system obtained from E by orientation of the equations[7]. The corresponding definition of satisfaction is:

$$M_P \models E \iff (\forall t \in \mathcal{T}_\Sigma,\ M_P \models t = t\downarrow)$$

$t\downarrow$ being a normal from, i.e. we consider the usual reflexive, symmetric and reflexive closure of the one-step rewriting relation, on ground terms.

This leads to the following exhaustive test set for a specification SP:

$$Exhaust_{TRS,SP} = \{t = t\downarrow \mid t \in \mathcal{T}_\Sigma\}$$

and the minimal hypothesis $Hmin_{TRS}$ is that M_P is a Σ-interpretation. It is possible to use the same patterns of testing hypotheses, i.e. uniformity and regularity, as for Example 1. One can remark that $Hmin_{TRS}$ does not require M_P to be finitely generated w.r.t. Σ. This is due to the way the satisfaction relation is defined.

The basic testing context is defined by $Hmin_{TRS}$, $Exhaust_{TRS,SP}$. Each $t\downarrow$ can be computed by an ASSPEGIQUE-like [20] or an OBJ-like system [21] and the oracle procedure is an equality decision, similar to the oracle of Example 1.

Let us consider now another example of institution which is particularly interesting since it nicely deals with the oracle problem.

Example 3 (observational semantics of algebraic specifications):
We consider here that the class of models of $SP = < \Sigma, E, Obs >$ is the class $Beh(SP)$ as defined by [22, 14, 23] and others. Obs is a subset of the sorts of SP, called the observable sorts.
The notion of *observable contexts* is crucial for observational semantics: an *observable context* over a sort s is a Σ-term C of observable sort, with exactly one occurrence of one variable of sort s. Then, a Σ-algebra A *satisfies* an equation $e = < L, R >$ iff $C[\overline{\theta}(L)] = C[\overline{\theta}(R)]$ for *all* assignements θ and *all* observable contexts C over the sort of e.

Then the observational exhaustive test set for a specification SP is:

$$Exhaust_{OBS,SP} = \{C[t] = C[t'] \mid t, t' \in s,\ (t = t') \in Exhaust_{ADJ,SP},\ C \textit{ observable context over } s\}$$

and $Hmin_{OBS} = Adequate_\Sigma$.

Remark:
Let us note that algebraic specifications with observational semantics is not an institution in general, as proved in [23], because the so-called "satisfaction condition" is not satisfied. However, in the particular case

[7] When this TRS is confluent and terminating, this ensures the existence of at least one model.

of program testing, this condition is fulfilled because the considered programs have a fixed observational signature which corresponds to the observable types and operations of the programming language. Anyway, the satisfaction condition is never used in our framework (we only need an explicit definition of the satisfaction relation).

This institution is interesting for our testing purpose since it provides a way of solving the oracle problem with weak hypotheses: let us take as Obs the set of predefined sorts (with equality) of the programming language. It is sensible to assume that these equality implementations are correct (however, it remains a hypothesis). Then the oracle decision just uses these equalities. This approach is reported in [11] and [12].

More generally, in order to apply our formal testing approach, the institution under consideration must provide a way of defining the exhaustive test set $Exhaust_{SP}$ as well as a way of characterizing a correct oracle *success*. Roughly speaking, the institution should provide us with a notion of "ground formulas" canonically derived from a set of axioms (the exhaustive test set), as well as a practicable definition of the satisfaction of ground atoms and some satisfaction rules associated with the connectives. Notice that this can be difficult, for instance if existential quantifiers are allowed in the specifications.

7 Testing and proving

One of the possible continuation of this work may be the definition of a verification framework where proving and testing would complement each other, in a way similar to proofs and refutations in mathematics [24]. A natural idea is to prove the hypotheses. It is not clear that it is always worth the effort: in Example 3, proving the oracle hypothesis can be very costly, and not truly necessary ; on the contrary, $Adequate_{\Sigma}$ can be proved easily if the programming language provides a well-defined encapsulation mechanism; some uniformity hypotheses could be verified by static analysis of the program. This point deserves further reflections: this implies to have a common theoretical background for testing and proving. Note that our ideas in this area are very tentative; they are reported in the following part of this paper for the purpose of opening a discussion.

A first obvious difference between proving and testing is that a proof works on the text of the program (let us call it P) and that a test exercises the system obtained from P, i.e. in our framework M_P. However, it is possible to exhibit some similarities. Let us consider a proof method i.e. some set of inference rules allowing to deduce facts of the form:

$$\vdash correctness(P, SP)$$

where $correctness(P, SP)$ is the so-called "correctness theorem", or the "proof obligation" associated with the development of P from SP [25], we can consider that it is an oracle. The set of properties required by SP (or a subset of it) can

be directly considered as a part of the test set occurring in our testing contexts. In this case, we assume some "Birkoff-like" hypothesis, namely :

$$\vdash correctness(P, \phi) \iff M_P \models \phi$$

This hypothesis means that it is assumed that both the correctness theorem and the execution support of P (i.e. translators, operating system) are consistent with the semantics M_P of P ... Similar issues are studied in the perspective of software proving in the ProCoS research project [26].

Conclusion

We propose a formalization of black-box testing, i.e. of those testing strategies which do not depend of the program structure, but on its specification only. This approach is based on formal specifications and it seems to be rather general and applicable to a significant class of formalisms. Moreover, we have introduced a notion of testing hypothesis which expresses the gap between the success of a test and correctness, and which could bring a way of combining testing and proving.

As soon as specifications are handled in a formal framework and the testing strategies are expressed as hypotheses (i.e. formulas), one can use proof-oriented tools for the selection of test data sets. These tools depend on the kind of formal specification in use: in [10] and [12] we present a tool based on "Horn clause logic" which allows one to deal with algebraic specifications and produces test sets corresponding to the combination of some general hypothesis schemes. Our tool is even more elaborated since it helps in the choice of hypotheses.

Acknowledgements
This work has been partially supported by the PRC "Programmation et Outils pour l'Intelligence Artificielle" and the ESPRIT basic research actions COMPASS and PDCS.

References

1. Proc. 1st ACM-IEEE Workshop on Software Testing, Banff, July 1986.
2. Proc. 2nd ACM-IEEE Workshop on Software Testing, Verification, and Analysis, Banff, July 1988.
3. Goodenough J.B., Gerhart S.L. *Towards a theory of test data selection.* IEEE trans. soft. Eng. SE-1, 2, 1975. Also: SIGPLAN Notices 10 (6), 1975.
4. Gourlay J.S. *A mathematical framework for the investigation of testing.* IEEE Transactions on Software Engineering, vol. SE-9, n^o 6, november 1983.
5. Marre B. *Génération automatique de jeux de tests, une solution : Spécifications algébriques et Programmation logique.* Proc.Programmation en Logique, Tregastel, CNET-Lannion, pp.213-236, May 1989.
6. Bougé L. *A proposition for a theory of testing: an abstract approach to the testing process.* Theoretical Computer Science 37, 1985.

7. Bougé L. Choquet N. Fribourg L. Gaudel M. C. *Test sets generation from algebraic specifications using logic programming.* Journal of Systems and Software Vol 6, n^o4, pp.343-360, November 1986.
8. Choquet N. *Test data generation using a PROLOG with constraints.* Workshop on Software Testing, Banff Canada, IEEE Catalog Number 86TH0144-6, pp.132-141, July 1986.
9. Gaudel M.-C., Marre B. *Algebraic specifications and software testing: theory and application.* LRI Report 407, Orsay, February 1988, and extended abstract in Proc. workshop on Software Testing, Banff, IEEE-ACM, July 1988.
10. Marre B. *Toward automatic test data set selection using Algebraic Specifications and Logic Programming.* Proc. Eighth International Conference ICLP'91 on Logic Programming, Paris, june 25-28, 1991.
11. Bernot G. *Testing against formal specifications : a theoretical view.* TAPSOFT CCPSD 91, LNCS 467, Brighton, April 1991.
12. Bernot G., Gaudel M.-C., Marre B. *Software Testing based on Formal Specifications: a theory and a tool.* Software Engineering Journal, to appear, november or december 1991.
13. Goguen J.A., Burstall R.M. *Introducing institutions.* Proc. Workshop on Logics of Programming, Springer-Verlag LNCS 164, pp.221-256,1984.
14. Hennicker R. *Observational implementation of algebraic specifications.* Acta Informatica, vol.28, n^{o}3, pp.187-230, 1991.
15. Weyuker E. J. *The oracle assumption of program testing.* Proc. 13th Hawaii Intl. Conf. Syst. Sciences 1, pp.44-49, 1980.
16. Weyuker E. J. *On testing non testable programs.* The Computer Journal 25, 4, pp.465-470, 1982.
17. Goguen J. Thatcher J. Wagner E. *An initial algebra approach to the specification, correctness, and implementation of abstract data types.* Current Trends in Programming Methodology, Vol.4, Yeh Ed. Prentice Hall, 1978.
18. Ehrig H., Kreowski H-J., Mahr B., Padawitz P. *Algebraic implementation of abstract data types.* Theoretical Computer Science, Vol.20, pp. 209-263, 1982.
19. Dauchy P., Marre B. *Test data selection from algebraic specifications: application to an automatic subway module.* ESEC'91, 3rd European Software Engineering Conference. Springer-Verlag (A.van Lamsweerde, A.Fugetta eds.) LNCS 550, pp. 80-100. Also: LRI report n^o 638, january 1991.
20. Bidoit M., Choppy C., Voisin F. *The ASSPEGIQUE specification environment, Motivations and design.* Proc. 3rd Workshop on Theory and Applications of Abstract data types, Bremen, Nov 1984, Recent Trends in Data Type Specification (H.-J. Kreowski ed.), Informatik-Fachberichte 116, Springer Verlag, Berlin-Heidelberg, pp.54-72, 1985.
21. Kirchner C., Kirchner H., Meseguer J. *Operational semantics of OBJ3.* Proc. 15th Int. Coll. on Automata, Languages and Programming (ICALP), Springer-Verlag, LNCS 317, pp.287-301, 1988.
22. Kamin S. *Final Data Types and Their Specification.* ACM Trans. on Programming Languages and Systems (5), pp.97-123, 1983.
23. Bernot G., Bidoit M. *Proving the correctness of algebraically specified software: Modularity and Observability issues.* In this conference AMAST-2, may 22-25, Iowa city, Iowa, USA, 1991.
24. Lakatos I. *Proofs and Refutations, The logic of mathematical discovery.* J.Worrall and E.Zahar ed., Cambridge Univ. Press, 1976.

25. Jones C. B. *Systematic software development using VDM.* Second edition, Comp. Sc. Series, Prentice Hall, C.A.R. Hoare ed., 1990.
26. Bjorner D. *A ProCos Project Description.* EATCS Bulletin, October 1989.

This article was processed using the LaTeX macro package with LLNCS style