Formal Methods and Testing: Hypotheses, and Correctness Approximations

Marie-Claude Gaudel

LRI, Paris-Sud University & CNRS, Orsay, France
mcg at lri.fr

Abstract. It has been recognised for a while that formal specifications can bring much to software testing. Numerous methods have been proposed for the derivation of test cases from various kinds of formal specifications, their submission, and verdict. All these methods rely upon some hypotheses on the system under test that formalise the gap between the success of a test campaign and the correctness of the system under test.

1 Introduction

It has been recognised for a while that formal specifications and models can bring much to software testing [16], [10]. In this extended abstract, we first precisely introduce the distinction between specification testing, model checking, and implementation testing based on formal specifications. Then we focus on the specificities of the latter one.

Actually, embedding implementation testing within a formal framework is far from being obvious. One tests a system. A system is a dynamic entity. It raises tricky issues such as observability and controllability, and sometimes specific physical constraints. A system is not a formula, even if it can be partially described as such. Thus, testing is very different from program proving, even if it is related. Similarly, testing is different from model checking, where verifications are performed on a known model: when testing, the model corresponding to the system under test is unknown (if it was known, testing would not be necessary…) and it is sometimes difficult to observe in what state it is [20], [22]. These points have been successfully circumvented in several testing methods based on formal specifications (or models) that use various and diverse techniques such as graph theory, symbolic evaluation, proof techniques, constraint solving, static analysis or model checking.

Explicitly or not, all these methods rely upon hypotheses on the system under test. They provide some approximation of correctness that is correlated to these hypotheses. In this talk we recall the notions of testability hypotheses and selection hypotheses that were introduced in [4], and we show how they have been used or could be used on various kinds of formal methods. We also address the issues of observation and control of the system under test.

2 Testing Specifications, Checking Models, or Testing Implementations?

Before starting some discussion on formal methods and testing, it is necessary to introduce some terminology. Unfortunately, there is no consensus on these issues among the various research communities working in the area of software.

There is not even an agreement on the meanings of the words "validation" and "verification" [7] [3]. Similarly, the word "testing" is often used with different meanings.

Looking in a dictionary, one gets definitions such as:

"subjecting somebody or something to challenging difficulties"

In the case of software and formal methods, the "somebody or something" and the "challenging difficulties" are sometimes understood in different ways.

In most cases, the entity under test is a system, and the "challenging difficulties" are inputs, or sequences of inputs, aiming at revealing some dysfunctions [4], [8], [12] [13], [15], etc. In such cases, formal descriptions of the system are mainly used as guidelines for the selection of (sequences of) inputs and for the verdict. We focus on these approaches in Sections 3, 4 and 5.

2.1 Debugging or Testing Formal Specifications

In some other cases, testing is understood as debugging of formal descriptions or models. The formal description is the subject of the test. The challenges are either properties to be satisfied or refuted [14], or inputs for some simulation of the future system, based on the formal description [19], [11].

As the main characteristic of formal specifications is the ability of reasoning, theorem proving is used either to prove that a required property is a consequence of the specification, or to refute a property that corresponds to a forbidden situation. The choice of such challenges is far from being simple. It requires a very good expertise in the application domain. As the specification may be wrong, is probably a good idea to make this choice as independent of it as possible [2], even if some positive experiments have been performed on mutation of formal specifications [6].

2.2 Checking Models … or Testing them?

Model checking is similar in purpose: it aims at finding faults in so-called models of software systems. These models are behavioural (Kripke Structures, Finite Automata, Finite State Machine, Labelled Transition Systems, or even program control graphs), with a finite (but often huge) number of states labelled by atomic propositions. A model checker checks properties written in some temporal logic via an exhaustive exploration of the model, or some equivalent technique. Here also, the choice of the temporal properties to be checked is far from being obvious.

Model checking could be seen as a special kind of testing where the subject is a model and the challenges are temporal properties. Actually, there is some evolution in

this direction. Due to the state explosion problem new techniques have been proposed that somewhat give up exhaustiveness: for instance, bounded model checking [5] where only finite prefixes of traces are considered; or randomised exploration of models until a target coverage quality is reached [17].

2.3 Testing Implementations

When testing implementations against a formal specification, the situation is different. As said in the introduction, the subject of the test is an executable system, whose internal state is often unknown. The system under test is not a formal entity. The only way to observe it is to interact via some specific (and often limited) interface, submitting inputs and collecting outputs.

3 Specifications, Implementations, and Testing

Given a specification *SP* and a system under test *SUT*, any testing activity must be based on a *relation of satisfaction* (sometimes called conformance relation) that we note *SUT sat SP*. This relation is usually defined on a semantic domain common to implementations and specifications (i.e. there is some domain *D* such that *sat* ⊆ *D*×*D*) [4], [9], [20], but in some cases they may be different (*sat* ⊆ *D1*×*D2*) [9].

3.1 Test Experiments, Exhaustiveness, and Testability

The satisfaction relation *SUT sat SP* is generally a large conjunction of elementary properties (for instance it may begin by "for all traces in the specification…"). These elementary properties are the basis for the definition of what is a *test experiment*, a test data, and the *verdict* of a test experiment, i.e. the decision whether *SUT* passes a test *t*. The satisfaction relation as a whole is used for the definition of an *exhaustive test set, Exhaust(SP)*.

However, an implementation's passing all the tests in the exhaustive test set does not necessarily mean that it satisfies the specification. This is true for a class of reasonable implementations. But a totally erratic system, or a diabolic one, may pass the exhaustive test set and then fail. More formally, the implementation under test must fulfil some basic requirements coming from the semantic domain considered for the implementations. As an example, in the case of finite state machines [20], the implementation must behave without memory of its history. Or when faced to non-deterministic *SUT*, some reasonable assumptions on the way of controlling it, or on the way of covering all the possible behaviours, are needed. We call such properties of the implementation the *testability hypothesis*, or the *minimal hypothesis*. We will note it *Hmin(SUT)*.

Hmin, *Exhaust*, and *sat* must satisfy:

$$Hmin(SUT) \Rightarrow (SUT \text{ passes } Exhaust(SP) \Rightarrow SUT \text{ sat } SP) . \quad (1)$$

There are cases where several choices are possible for the pair $<Hmin, Exhaust>$. When restricting the class of implementations under test, using for instance some knowledge on the way it was developed, it is possible to lessen *Exhaust(SP)*.

3.2 Selection Hypotheses, Uniformity, Regularity

A black-box testing strategy can be formalised as the selection of a finite subset of *Exhaust(SP)*. Let us consider as an example the classical partition testing strategy (more exactly, it should be called sub-domain testing strategy). It consists in defining a collection of (possibly non-disjoint) subsets that covers the exhaustive test set. Then a representative element of each subset is selected and submitted to the implementation under test.

The choice of such a strategy corresponds to stronger hypotheses than *Hmin* on the system under test. We call such hypotheses *selection hypotheses*. In this case, it is a *uniformity hypothesis*. The system is assumed to uniformly behave on the test subsets UTS_i:

$$UTS_1 \cup \ldots \cup UTS_p = Exhaust(SP), \quad and$$

$$\forall i = 1, \ldots, p, \forall t \in UTS_i, SUT \text{ passes } t \Rightarrow SUT \text{ passes } UTS_i \quad (2)$$

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. A *regularity hypothesis* uses a size function on the tests and has the form "if the subset of *Exhaust(SP)* made up of all the tests of size less than or equal to a given limit is passed, then *Exhaust(SP)* also is" (there is some similarity with bounded model checking).

All these hypotheses are important from a theoretical point of view because they 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. It gives some indication of complementary verifications.

Weak selection hypotheses lead, via formula (1), to large test sets. Strong selection hypotheses lead to smaller, more practicable test sets, with the risk that they may not be fulfilled. The strongest selection hypothesis is the correctness assumption: in this case, an empty test set is sufficient…

There exist various ways to select test sets in the framework of specification-based testing. The most used are coverage criteria based on the specification. A well-known example in the case of finite state machines is transition coverage [10]. It corresponds to a testability hypothesis that the SUT is some deterministic FSM. Another approach is to select tests via a finite number of test purposes describing some behaviours that are considered to be important to test. Combining the specification and the tests purposes, a finite number of test cases are generated. This kind of selection is

used for example in the TGV tool [9]. It can be formalised as some restriction of the conformance relation combined with some selection hypotheses.

3.3 The Oracle Problem

The interpretation of the results of a test is often very difficult. This difficulty is known as the *oracle problem*. The problem may be difficult for various causes.

The *SUT* may yield the results in a way that depends on some representation choices and makes the comparison with the specified results difficult. The test is based on a specification that is (normally) more abstract than the program. Thus program results may appear in a form that is not obviously equivalent to the specified results. This contradicts a common belief that the existence of a formal specification is sufficient to directly decide whether a test is a success. In presence of complex data types, it may be necessary to embed the tests into observable contexts, or to enrich the *SUT* with some concrete equivalence function [22].

Similarly, when the specification is based on states and transitions, it may be difficult to check that the *SUT* is in an acceptable state after a test. It may require complementing the test itself by some other tests for identifying the internal state [20].

4 Axioms, Pre-conditions and Post-conditions

Historically the above framework has been developed for algebraic specifications [4], [22]. Test data are just instantiated axioms of the specification and test experiments consist in their evaluation by the *SUT* to check that they are satisfied. The exhaustive test set is the set of all closed instances of the axioms of the specification. The testability hypothesis on the *SUT* is that all the functions of the signature are implemented in a deterministic way, and that there is no junk (no unspecified values). A basic testing strategy is to cover once every axiom. It corresponds to uniformity hypotheses on the domains of their variables. This strategy can be refined by composing axioms (unfolding functions) in order to get a better coverage of sub-cases, i.e. weaker uniformity hypotheses. In the case of positive conditional axioms, this method has been automated by the LOFT constraint solver [4].

In the case of VDM, Jeremy Dick and Alain Faivre [12] have proposed to reduce the pre conditions and post conditions into disjunctive normal forms (DNF), creating a set of disjoint input sub-domains for each operation of the specification. This provides a nice way of discovering uniformity hypotheses. As VDM is state-based, it is not enough to partition operations domains: thus the authors give a method of extracting a finite state automaton from the specification. It uses the uniformity sub-domains of the operations to perform a partition of the states. Given this finite state automaton one can use one of the testing methods mentioned in the next section. This work has been influential on several researches on testing based on formal methods close to VDM, such as Z, or B, that are too numerous to be all cited here.

More recently, similar ideas have been used in the KORAT framework for testing Java methods specified by JML preconditions and post conditions [8]. KORAT derives from the precondition "all non isomorphic test cases up to a given small size", i.e. the selection is based on a combination of uniformity and regularity hypotheses.

5 Behavioural Models, FSM, LTS, etc

Historically, finite state machines (FSM) have been the first formal descriptions used as basis for automatic test derivation [10]. Originally, there was a testability hypothesis that the SUT behaves as a FSM with the same number (or a larger known number) of states as the specification FSM. The conformance relation was equivalence. These choices were adequate for hardware testing, which was the original motivation. The excellent survey by Lee and Yannakakis presents extensions to more elaborated conformance notions, and to extended state machines [20]. Similar approaches have been developed in the area of communication protocols, based on labelled transition systems (LTS) or variants of them [9]. In [15] and [21] we have stated the underlying notions of testability hypotheses, exhaustive test sets, and selection hypotheses for these approaches.

For some years, there is a fruitful cross-fertilisation between these so-called model-based testing methods and model checking techniques (cf. [1] [18], [23] among many others). For instance, the ability of model checker to provide counterexamples can be used to produce test sequences that satisfy a property *P* by model-checking the property *"always not P"*. Model checkers are now among the major tools for testing based on formal specification, together with constraint solvers, theorem provers, and symbolic interpreters.

6 Conclusion

There has been a lot of work on test cases derivation from formal descriptions. It is our claim that formal approaches bring more than that to testing. They make it possible to state the underlying hypotheses associated with test strategies and thus to express the correctness approximation they introduce. This open a lot of possibilities, first for identifying complementary verifications, second for assessing these approximations.

References

1. Ammann, P. E., Black, P.E., Majurski, W.⊡ Using model checking to generate tests from specifications. IEEE International Conference on Formal Engineering Methods (ICFEM'98), IEEE , (1998) 46-54.

2. Arnold, A., Gaudel, M.-C., Marre B.: An experiment on the validation of a specification by heterogeneous formal means. 5th IFIP working conference on Dependable Computing for Critical Applications, Urbana Champaign, (1995) 24-34.
3. Avizienis, A., Laprie, J-C., Landwehr, C., Randell, B.: Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Trans. on Dependable and Secure Computing, vol. 1, n° 1, (2004) 11-33.
4. 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, (1991) 387-405.
5. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.□ Symbolic model checking without BDDs. TACAS'99, LNCS n° 1579, Springer-Verlag (1999) 193–207
6. Black, P.E., Okun, V., Yesha, Y.□ Mutation Operators for Specifications. IEEE International Conference on Automated Software Engineering (ASE2000), IEEE (2000) 81-88.
7. Boehm, B. W.: Software Engineering Economics, Prentice Hall (1981).
8. Boyapati, C., Khurshid, S., Marinov, D.: KORAT: automated testing based on Java predicates. ACM International Symposium on Software Testing and Analysis, (2002) 123-133.
9. Brinksma, E., Tretmans, J.: Testing Transition Systems, an annotated bibliography. Lecture Notes in Computer Science n° 2067, Springer-Verlag (2001) 187-195.
10. Chow, T. S.: Testing Software Design Modeled by Finite-State Machines. IEEE Transactions on Software Engineering, vol. SE-4, n° 3, (1978) 178-187.
11. Desovski, D.: Combining Testing and Model Checking for Verification of High Assurance Systems. IEEE Int. Symp. on High Assurance Software Engineering, IEEE (2004).
12. Dick, J., Faivre, A.: Automating the Generation and Sequencing of test cases from model-based specifications. International Symposium of Formal Methods Europe, Lecture Notes in Computer Science n°670, Springer-Verlag (1993) 268-284.
13. Farchi, E., Hartman, A., Pinter, S. S.: Using a model-based test generator to test for standard conformance. IBM Systems Journal, vol. 41, n° 1, (2002) 89-110.
14. Garland, S.J, Guttag, J.V.: Using LP to Debug Specifications. IFIP TC2 Working Conference on Programming Concepts and Methods, North-Holland (1990).
15. Gaudel, M.-C., James, P. R.: Testing Algebraic Data Types and Processes : a unifying theory. Formal Aspects of Computing, 10(5-6), (1999) 436-451.
16. Goodenough, J. B., Gerhart, S.: Toward a Theory of Test Data Selection. IEEE Transactions on Software Engineering, vol. SE-1, n° 2, (1975) 156-173.
17. Grosu, R., Smolka, S. A.: Monte Carlo Model Checking. TACAS 2005, Lecture Notes in Computer Science n° 3440, Springer-Verlag, (2005) 271–286.
18. Hamon, G., de Moura, L, Rushby, J.: Generating Efficient Test Sets with a Model Checker. IEEE Int. Conf. on Software Engineering and Formal Methods, IEEE, (2004) 261-270.
19. Kemmerer, R.A.: Testing Formal Specifications to Detect Design Errors. IEEE Transactions on Software Engineering, vol. SE-11, no 1 (1985) 32-43.
20. Lee, D, Yannakakis, M.: Principles and methods of Testing Finite State Machines – a survey. The Proceedings of IEEE, vol. 84, n° 8, (1996)1089-1123.
21. Lestiennes, G., Gaudel, M.-C.: Testing Processes from Formal Specifications with Inputs, Outputs, and Data Types. 13th IEEE Int. Symp. on Software Reliability Engineering (ISSRE-2002), IEEE, (2002) 3-14.
22. Machado, P. D. L,: On Oracles for Interpreting Test Results against Algebraic Specifications. Lecture Notes in Computer Science n° 1548, Springer-Verlag (1998) 502-518.
23. Peled, D., Vardi, M., Yannakakis, M.: Black Box Checking. Proceedings of FORTE/PSTV, Kluwer (1999) 225-240.