Formal Methods for Software Testing

Marie-Claude Gaudel

▶ **To cite this version:**

Marie-Claude Gaudel. Formal Methods for Software Testing. 11th International Symposium on Theoretical Aspects of Software Engineering (TASE 2017), , IEEE, Sep 2017, Sophia-Antipolis, France. hal-01683611

HAL Id: hal-01683611
https://hal.science/hal-01683611

Submitted on 14 Jan 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Formal Methods for Software Testing

(Invited Paper)

Marie-Claude Gaudel
LRI, Univ. Paris-Sud, CNRS, CentraleSupélec
Université Paris-Saclay
Orsay, France 91405
Email: mcg@lri.fr

***Abstract*—This extended abstract takes advantage of a theory of software testing based on formal specifications to point out the benefits and limits of the use of formal methods to this end.**

A notion of exhaustive test set is defined according to the semantics of the formal notation, the considered conformance relation, and some testability hypotheses on the system under test. This gives a framework for the formalisation of test selection, test execution, and oracles, and, moreover, leads to the explicitation of those hypotheses underlying test selection strategies, such as uniformity hypotheses or regularity hypotheses. This explicitation provides some guides to complementary proofs, or tests, or instrumentations of the system under test.

This approach has been applied to various formalisms: axiomatic specifications of data types, model-based specifications, process algebras, transition systems, etc. It provides some guiding principles for the development of testing methods given a formal specification notation and an associated conformance/refinement relation. It is at the origin of the development of some test environments based on SMT solvers and theorem provers.

I. Introduction

Testing software systems is a complex and difficult task, due to the diversity of the potential faults in such systems. Among several complementary approaches, specification-based software testing was perceived as unavoidable since the very first studies and experiments. Moreover, the links between software testing and formal specifications have been studied for quite a while: some pioneering papers come back to the seventies (see for instance [1] and [2]). Sound and effective testing methods have been established based on various types of formal specifications, leading to tools for assisting the testing process. A comprehensive survey can be found in [3]. All these works refer, explicitly or not, to the same formal concepts.

Taking advantage of a theoretical framework for software testing based on formal specifications, this paper points out and discusses the benefits and limits of the use of formal methods to this end. This framework was first developed for algebraic specifications [4], [5]. It has been instantiated for several formal approaches, such as VDM [6], finite state machines (FSM) [2], [7], various kinds of labelled transition systems (LTS) [8], Petri nets [9] and several process algebras: LOTOS [10], [11], CSP [12], *Circus* [13]. The emergence of powerful SMT solvers, such as Z3 [14], advances in symbolic execution [15], and the availability of proof assistants, such as Isabelle/HOL [16] have allowed the development of test environments such as HOL/TestGen [17], which permits synergies between tests and proofs.

II. Background: Specification-based Testing, Formal Methods.

A. Some Basic Facts on Specification-Based Testing

Specification-based testing, or model-based testing, are black-box testing methods, where the internal organisation of the system under test (SUT) is ignored: the test strategy is based on some description of the intended properties and behaviour of the SUT that can be formal or not.

A common-sense observation is that, being based on some specification, such methods are relevant for detecting those faults that cause deviations with respect to this specification, no more. For other kinds of fault, other validation and verification methods must be used. For instance, problems related to the execution support, such as overflows, are not caught by specification-based testing unless they are mentioned in the specification. Similarly, program-based methods are pertinent for discovering errors occurring in programs and not for omissions with respect to specifications.

Another observation is that one tests a system. A system is neither a formula, nor a diagram, even if it can be (partially) described as such. It is a dynamic entity, embedded in the physical world. It is observable via some limited interface or procedure. It is not always fully controllable. A specification, or a model, or a program text, are not the system but some description of the system. "The map is not the territory" [18]. Actually, when testing, some assumptions are made on the SUT: implicitly or explicitly, one considers a class of testable implementations. These assumptions on the SUT are called *testability hypotheses*. They may correspond to very basic assumptions such as correct implementation of booleans and bounded integers or determinism, but depending on the kind of considered specifications they may be more sophisticated as it will be discussed in Section III-A.

Saying it in another way, these methods target some classes of faults, assuming either that the SUT is exempt from other kinds of faults, or that other complementary methods are used. If the SUT is any erratic or diabolic system, there is no way of testing it with sensible outcome on the basis of abstract descriptions. This is the reason why binary code analysis (see for instance [19]) is used for detecting malicious faults.

B. Formal Methods in a Nutshell

What makes a specification method formal? As for any specification method, there is a notation. Depending on the

method, specifications can include formulas in various logics, used to write pre- post-conditions, axioms of data types, guards, temporal properties. They may state process definitions as for instance in CSP, CCS, Lotos, *Circus*. They may rely on annotated diagrams such as FSM, LTS, Petri nets, etc.

But there is more than a syntax. First, there is a formal semantics, in term of mathematical notions such as: predicate transformers for pre- post-conditions; sets and many-sorted algebras for axiomatic definitions; various sorts of automata, traces, failures, divergences, for process algebras. Second, there is a formal deduction system, making it possible to perform proofs, or other verifications (such as model-checking), or both. Thus formal specifications can be analysed to guide the identification of appropriate test cases.

Moreover, in addition to syntax, semantics, and deduction system, formal methods come with some relations between specifications that formalise either equivalence or correct stepwise development. Depending on the context such relations are called: refinement, conformance, or, in the case of formulas, satisfaction. They are fundamental for testing methods.

III. Formal Specifications and Testing

A. Bridging the Gap between Test and Refinement Relations

Given a formal specification and a system under test, the testing activity addresses the question: "does the SUT conform to the specification? ". However, embedding testing activities within a formal framework is not so straightforward. As said above, one tests a system while refinement/conformance relations are defined on pairs of formal descriptions: for instance, in the case of FSM, it is equivalence of FSM; in the case of CSP, traces refinement requires that the set of traces of the refined specification is a subset of the traces of the original one, and similarly for failure refinement, ...

The gap between systems and specifications is generally taken into account by testability hypotheses on the systems under test [2], [4], [7], called test hypotheses in [8].

For instance, in the case of FSM, it is required that the SUT behaves like some FSM with the same number of states as the specified one (or more, but this number is known). It means that whatever the trace leading to some state, the execution of a transition from this state with a given input has the same effect in term of output and change of state. This provides justification for the transition coverage testing strategy that is classical in this framework [7].

In the case of CSP, the SUT is assumed to behave like some unknown CSP process with the same alphabet of actions as the specified one, and that these actions are atomic [12] or are perceived as atomic and of irrelevant duration in the SUT. It is possible to ensure this requirement by developing wrappers, or by performing some complementary proofs or tests.

In summary, the system under test is assumed to behave like some (unknown) model of the same nature as the ones considered by the refinement/conformance relation. Such hypotheses are formal counterparts of the remarks in II-A.

B. Test, and Exhaustive Test Sets

Thanks to the formal deduction system and the refinement/conformance relation, one can derive consequences and counter-examples from the specification. They provide guides for designing testing experiments, for selection of test cases, and for specification of test drivers.

For instance: testing that a commutativity axiom such as $insert(t,x,y) = insert(t,y,x)$ is satisfied by the implementation of a table data type can be done by assigning some values to the variables t, x, y, and checking that the execution of both sides of the equation yield similar results; testing that an implementation of a CSP process conforms to traces refinement can be done by synchronising it with a trace that is not a trace of the process and checking that a deadlock is observed when the submitted trace deviates from the specification.

Due to dependencies between counter-examples or consequences it is not always necessary to consider all of them: for instance, in the case of traces refinements, instead of the set of all unspecified traces, it is sufficient to consider the minimal forbidden prefixes of these traces without loss of exhaustivity.

Exhaustivity of a test set TS with respect to a specification SP is the fact that a SUT passes all the tests in TS if and only if it behaves like a refinement of SP. It must take into account the testability hypotheses. It can be summarised as:
$Testable(SUT) \Rightarrow$
$\big((\forall t \in TS, SUT\ passes\ t) \iff ([|SUT|]\ refines\ SP)\big)$
where $([|SUT|]$ denotes the unknown formal specification that corresponds to SUT.

Examples of such theorems are given for algebraic specifications in [5], for CSP in [12], for *Circus* in [13]. For a given formalism, the definition of tests, and of the couple of exhaustive test set and testability hypotheses is not unique. For instance, strengthening the testability hypotheses makes it possible to reduce the exhaustive test set.

C. Selection Hypotheses

Once defined an exhaustive test set for a specification SP and the corresponding testability hypotheses on a system under test SUT, a specification-based testing strategy of SUT against SP can be formalised as the selection of a finite subset of the exhaustive test set and some strengthening of the testability hypotheses. We call the additional hypotheses *selection hypotheses*. They make explicit the incompleteness of practical testing methods.

As an example, let us consider the classical partition testing strategy. Given a collection of (possibly non-disjoint) subsets that covers the exhaustive test set, a representative element of each subset is selected to be submitted to the SUT. Let the following decomposition of a test set TS into n subsets $STS_1, ..., STS_n$, such that $TS = STS_1 \bigcup ... \bigcup STS_n$. The partition testing strategy corresponds to *uniformity hypotheses* on the subsets that can be expressed as:

$(\exists t_0 \in TS, SUT\ passes\ t_0) \Rightarrow (\forall t \in TS, SUT\ passes\ t)$
Other examples of selection hypotheses are *regularity hypotheses*, that allow to bound the size of the tests, and thus their

number. These are the two main classes of test selection hypotheses. They allow to specify coverage criteria of the specification: for instance the popular transition coverage criterion of FSM [2] can be formulated as a uniformity hypotheses on those subsets of traces where a given transition occurs. But depending on the underlying formalism and the considered class of faults, they are many variants. Regularity hypotheses can be combined with uniform drawing from the exhaustive test set following the ideas in [20].

As pointed out in [21], [22], test hypotheses, i.e. testability and selection hypotheses, are equivalent to fault domains, at least for black-box techniques: fault domains limit the set of considered SUT by restricting the class of targeted faults and it is exactly the role of test hypotheses.

There are several interesting consequences in formulating test strategies as explicit logical hypotheses. First, as developed in [21], it provides logical bases for comparing test criteria and identifying redundancies. Second, as illustrated in the HOL/TestGen test environment [17], it makes it possible to use proof assistants and SMT solvers to automate test selection and submission. HOL/TesGen is based on the powerful Isabelle/HOL proof assistant. It allows to specify or discover some uniformity and regularity hypotheses, to prove them, and to generate the corresponding test sets and drivers.

D. Oracles

The oracle problem, i.e. how to decide whether a test execution yields satisfactory outcome, is a very difficult issue in software testing [23]. The main causes of this difficulty are the lacks of observability and of controllability of the SUT.

In the case of specification-based testing, as soon as there is some abstraction gap between the abstract specification and the concrete implementation, problems arise for interpreting the results of test experiments: they are observed from the system, they must be interpreted w.r.t. the specification.

In formal methods that support abstract descriptions of complex data types, such as algebraic specifications, VDM, Z, this problem shows up for deciding the equivalence of different representations of the same abstract entity. For instance, testing the commutativity of insertion in a table (cf. the axiom in III-B) is problematic when using hashing. An elegant formal way to cope with this issue is to wrap the tests in *observation contexts* to get results of observable types, i. e. in term of the implementation [5], [24], [25]. From the specification, one identifies sequences of operations from abstract types to simple types treatable by the SUT. It is similar to what is done for state identification of the SUT in the case of FSM-based testing some characterising set of sequences is built from the FSM [2], or some distinguishing sequences [7]. Such sequences when applied to different states yield different outputs. Appended to tests they indicate the resulting state.

These methods are expensive, and in some cases incomplete. Other, less formal, grey-box approaches relies on instrumentations of the SUT in order to improve its observability [26], raising the issue of the correctness of this instrumentation that may be proved or tested w.r.t. the formal specification.

IV. Conclusion

Formal methods are one important ingredient of a holistic approach to software testing. They are not the "silver bullet" but they bring much, thanks to their logical background. One of their essential advantages is the explicitation of the assumptions on the SUT associated with testing strategies, thus paving the way to complementary proofs or tests.

References

[1] J. B. Goodenough and S. L. Gerhart, *Toward a theory of test data selection*, IEEE Trans. on Software Engineering, SE-1(2): 156-173, 1975.

[2] T. Chow, *Testing software design modeled by finite-state machines*, IEEE Trans. on Software Engineering, SE-4(3):178187, 1978.

[3] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Luttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan, *Using formal specifications to support testing*, ACM Comput. Surv. 41(2), 1-76, 2009.

[4] G. Bernot, M.-C. Gaudel, and B. Marre, *Software testing based on formal specifications: a theory and a tool*, Soft. Eng. Journal, 6(6):387-405, 1991.

[5] M.-C. Gaudel and P. Le Gall, *Testing data types implementations from algebraic specifications*, LNCS 4949, 209-239, 2007.

[6] J. Dick and A. Faivre, *Automating the generation and sequencing of test cases from model-based specifications*, LNCS 670, pp. 268-284, 1993.

[7] D. Lee and M. Yannakakis, *Principles and methods of testing finite state machines-a survey*, Proceedings of the IEEE, 84(8):1090-1123, 1996.

[8] E. Brinksma and J. Tretmans, *Testing Transition Systems: An Annotated Bibliography*, LNCS 2076, 187-195, 2001.

[9] C. Péraire, S. Barbey and Didier Buchs, *Test selection for object-oriented software based on formal specifications*, IFIP TC2/WG2.2,2.3 PROCOMET Conference, 385-403, 1998.

[10] D. H. Pitt and D.Freestone, *The Derivation of Conformance Tests from LOTOS Specifications*, IEEE Trans. on Software Engineering, SE-16(12):1337-1343, 1978.

[11] M.-C. Gaudel and P. R. James, *Testing Algebraic Data Types and Processes: a unifying theory*, FACJ, 10(5-6), 436-451, 1999.

[12] A. Cavalcanti and M.-C. Gaudel, *Testing for refinement in CSP*, LNCS 4789, 151-170, 2007.

[13] A. Cavalcanti and M.-C. Gaudel, *Testing for refinement in Circus*, Acta Informatica, 48(2):97-147, 2011.

[14] L. De Moura and N. Bjørner. *Z3: An efficient SMT solver*, LNCS 4963, 337-340, 2008.

[15] C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, K. Sen, N.Tillmann, and W. Visser, *Symbolic execution for software testing in practice: preliminary assessment*, ICSE '11, 1066-1071, 2011.

[16] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, LNCS 2283, 2002.

[17] A. D. Brucker and B. Wolff, *On Theorem Prover-based Testing*, Formal Aspects of Computing, 25(5):683-721, 2013.

[18] A. Korzybski, *Science and Sanity: A Non-Aristotelian System and its Necessity for Rigour in Mathematics and Physics* Institute of General Semantics,1933.

[19] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, A. Vincent, *The BINCOA framework for binary code analysis*, LNCS 6806, 165170, 2011.

[20] A. Denise, M.-C. Gaudel, S.-D. Gouraud, R. Lassaigne, J. Oudinet and S. Peyronnet, *Coverage-biased random exploration of large models and application to testing*, STTT 14(1): 73-93, 2012.

[21] R.M. Hierons, *Comparing test sets and criteria in the presence of test hypotheses and fault domains*, ACM TOSEM 11(4): 427-448, 2002.

[22] R. M. Hierons, *Verdict functions in testing with a fault domain or test hypotheses*, ACM TOSEM, 18(4): 14:1-14:19, 2009.

[23] E. T. Barr, M. Harman, Ph. McMinn, M. Shahbaz and Shin Yoo, *The Oracle Problem in Software Testing: A Survey*, IEEE Trans.on Software Engineering, 41(5), 507-525, 2015.

[24] H. Y. Chen, T. H. Tse and T. Y. Chen, *TACCLE: a methodology for object-oriented software testing at the class and cluster levels*, ACM TOSEM 10(1), 56-109, 2001.

[25] H. Zhu, *A note on test oracles and semantics of algebraic specifications*, IEEE QSIC conference, 91-99 2003.

[26] P. Machado, *On oracles for interpreting test results against algebraic specifications*, LNCS 1548, 502518, 1998.