Volume 3, Issue 2, February 2013 ISSN: 2277 128X

International Journal of Advanced Research in Computer Science and Software Engineering

Research Paper

Available online at: www.ijarcsse.com

Formal methods: A Complementary Support for Testing

Monika Singh

Assistant Professor,
Mody Institute of Technology and Science (MITS),
Lakshmanghar,
Rajasthan, India

Abstract---Formal methods and testing are two important approaches, which are useful in developing high quality software. Formal methods have traditionally been used for specification and development of software. While traditionally these approaches i.e. formal methods and testing, have been seen as rivals, but in recent years a new consensus has developed in which they are seen as complementary. However there are potential benefits for the testing stage as well. This article focuses on the usefulness of formal methods in assisting software testing in order to improving software quality and reduces the cost.

Keywords--- Formal specification languages, SFT (system for testing), FSM, VDM, Z.

I. Introduction to Formal methods

Formal specification languages are mathematically based languages whose purpose is to aid the construction of systems and software. As formal methods are supported by tool, they can be used to both describe a system and also then to analyze its functionality, possibly verifying key characteristics of interest. The engineering constraints behind formal methods are that time spent on specification [1] [4]. Of course formal methods do not guarantee correctness, but their use emphasis to increase understanding of a system by revealing errors or aspects of incompleteness that might be expensive to correct them at any later point of time. The development cycle is still likely to be iterative, because the software requirements usually change during the development of a project and the use of formal specification languages cannot be expected to entirely eliminate errors.

II. Formal methods and testing

Formal methods and testing is sometimes seen competitor. It has been said that formal methods could eliminate testing. In fact, formal methods and testing will always be two complementary techniques for the reduction of errors in computer-based systems. The use of a formal specification or model eliminates ambiguity and thus reduces the chance of errors being introduced during software development. Unfortunately, to choose a formal specification language that matches the actual customer requirements is still an issue and this is complicated by the tendency for stated requirements to change during development [6] [7] [10]. Where a formal specification exists, both the source code and the specification may be seen as formal objects that can be analyzed and manipulated. If this is done then we have greater confidence that we are testing the system for testing (SFT*)* against the actual requirements. By using formal specification, we can introduce the possibility of the formal and automatic analysis of the relationship between the software specification and the source code. Generally we assumed to take the form of a proof, but such a proof cannot guarantee the correctness of operations. Therefore in a situation, even where such a proof is likely to exist, it would be more significant to apply dynamic testing. By using formal methods and testing together, we can reduce the development cost by applying testing techniques in earlier phases of the lifecycle where defects are relatively inexpensive to correct.

III. Testing from Formal specification languages

A variety of different formal specification techniques exist, in which some are general purpose while others are useful for a particular application domains (e.g., concurrent systems). The following are the most popular formal notations and how their use can be integrated into the testing process [11] [15] [16].

Model-Based Languages

There are a number of different approaches to write an unambiguous and correct specification. One way is to build a *model* of the intended behavior, and languages such as Z [Spivey 1988; 1992], VDM [Jones 1991] and B [Abrial 1996] do so by describing the states the system could be in together with operations that change the state. Model-based languages such as Z, VDM and B are used to describe arbitrarily general systems, and have potentially infinite state. In case of Model-based language, the test generation techniques are typically based on the uniformity hypothesis: they partition the input domain into

a set of sub domains on which the behavior of the specification is uniform [3]. Test cases are then generated from each sub domain, and potentially around the boundaries of the sub domains. This process can be seen in terms of one of two test hypotheses. One test hypothesis is that if the SFT is faulty then its behavior is equivalent to that of one of the mutants. An alternative test hypothesis is that if a test set distinguishes between the specification and mutants then it distinguishes between the specification and any faulty SFT [7].

Finite State-Based Languages

Finite state-based languages define their state from a finite set of values, which are generally presented graphically with state transitions representing changes of state similar to operations in Z notation. Examples of such languages include finite state machines (FSMs) [Lee and Yannakakis 1996], SDL [ITU-T 1999], Statecharts [Harel and Gery 1997] and X-machines [Holcombe and Ipate 1998]. One should consider the FSM based test techniques when testing from such specifications. In particular FSM used in specifying communications protocols. Most work in the area of testing from finite state-based models does not discuss test hypotheses but instead considers fault models [8]. However, fault models are an extremely similar concept: like test hypotheses they postulate restrictions on the behavior of the SFT and are used to drive test generation. In fact, some of the standard assumptions made in FSM based testing are similar to the test hypotheses. For example, the assumption when testing from an FSM with n states that the SFT behaves like an unknown FSM with at most m states can be seen as a regularity hypothesis: if this holds and the SFT is faulty then there exists an input sequence of length at most $m+n+1$ that leads to a failure. The lack of expressiveness of FSMs but their suitability for automated test sequence generation has led to situations in which developers and testers write specification or models in more expressive languages and a tool converts these into FSMs that then act as the basis for test sequence generation. While this seems to provide the benefits of both expressive languages and automated test sequence generation, a number of challenges remain [9] [13]. For many systems we cannot simply produce an FSM by taking all combinations of values for internal variables and all states: this may lead to an infinite state space and even if the state space is finite it is likely to be extremely large. If we apply an abstraction then we have two issues to consider. First, this may abstract out information that could help us in testing. Second, the abstraction could lead to a model in which there are paths that are feasible and so could be chosen in test sequence generation, but do not correspond to feasible paths in the original model.

Process Algebra State-Based Languages

Concurrency of a system can be understood as a number of communicating concurrent processes by applying classic algebraic treatment, and process algebras. Examples include CSP [Hoare 1985], CCS [Milner 1989] and LOTOS [ISO 1989a]. Finite-state based languages such as State charts and SDL can also be used to describe a system as a set of communicating concurrent processes. In spite of, process algebras have a rich theory that provides alternative notions of conformance described in terms of implementation relations, the implementation relations' acquirement several types of observations that can be made, in addition to traces, and different properties of the environment. Process Algebras and FSMs are similar in that they model or specify a system through a set of states and transitions between these states [12] [14]. Process Algebras are the more expressive, especially when dealing with non-determinism, and the implementation relations relate to the ability of the tester to make observations. Test hypotheses and fault models have played a relatively small role in the literature on testing from a process algebra [5]. It is sometimes possible to apply an alternative approach to testing from an LTS which involves converting the LTS specification into an FSM. A test suite is then generated from this FSM (see, for example, [Petrenko et al. 1994]). The advantage of this approach is the ability to use the fault models and test generation algorithms developed for FSMs. However, there is the disadvantage that either we are restricted to the implementation relations that are used in testing from an FSM or we have to produce a different transformation for each implementation relation.

Hybrid Languages

Many systems are made of with a combination of analog and digital components. It is necessary to use a specification language that encompasses both discrete and continuous mathematics to specify and verify such systems. There has been recent interest in these *hybrid languages*, such as CHARON [Alur et al. 2000]. The inclusion of continuous variables adds significant challenges and makes the application of testing approaches based on the state structure problematic. It appears that the main focus has been on the development, analysis and simulation of hybrid automata; testing is achieved through simulation. The situation is rather different for a special type of hybrid automaton: timed automata. Here there is only one continuous variable, time, possibly represented by clocks. By defining an equivalence relation on time it is possible to partition the states of the timed automaton and form a finite automaton from which test sequences can be produced [2] [11]. FSM test hypotheses and techniques can be used with timed automata; it is unclear how useful these are in practice. First, there is the possible state explosion when partitioning the states of the timed automaton. By using a coarser equivalence relation we can get fewer states but naturally the corresponding test sequences `cover' less of the model. Currently there appears to be a lack of evidence regarding the trade-off between the cost of testing and test effectiveness as influenced by the choice of equivalence relation used. In addition, the test hypotheses used concern the automaton produced by state partitioning but it may be more meaningful, for the tester, to phrase test hypotheses in terms of the timed automaton model.

IV. Conclusion

By this article I try to describe how the formal specification can be used in order to assist testing. We have seen that software testing benefits from the presence of a formal specification in a number of important ways, particularly, the presence of a formal specification aids test automation and allows the tester to reason about test effectiveness.

References

[1]. AL-AMAYREH, A. AND ZIN, A. M. 1999. PROBE: A formal specification-based testing system in *20th International conference on Information Systems*. 400.404.

[2]. ANTSAKLIS, P., KOHN, W., NERODE, A., AND SASTRY, S., Eds. 1995. *Hybrid Systems II*. Lecture Notes in Computer Science, vol. 999. Springer-Verlag.

[3]. BEHNIA, S. AND WAESELYNCK, H. 1999. Test criteria definition for B models. In *World Congress on Formal Methods*. 509.529.

[4]. BERNOT, G., GAUDEL, M.-C., AND MARRE, B. 1991. Software testing based on formal specifications: A theory and a tool. *IEE/BCS Software Engineering Journal 6*, 6, 387.405.

[5]. BOUG´E, L., CHOQUET, N., FRIBOURG, L., AND GAUDEL, M.-C. 1986. Test sets generation from algebraic specifications using logic programming. *The Journal of Systems and Software 6*, 4, 343.360.

[6]. CARRINGTON, D. A. AND STOCKS, P. A. 1994. A tale of two paradigms: Formal methods and software testing. In *Z User Workshop, Cambridge 1994*, J. P. Bowen and J. A. Hall, Eds. Workshops in Computing. Springer-Verlag, 51.68.

[7]. CIANCARINI, P., CIMATO, S., AND MASCOLO, C. 1996. Engineering formal requirements: Analysis and testing. In *8th International Conference on Software Engineering and Knowledge Engineering (SEKE), Lake Tahoe*. 385.392.

[8]. Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Pasareanu, C. S., Robby, and Zheng, H. 2000. Bandera: extracting finite-state models from Java source code. In *International Conference on Software Engineering*. IEEE Computer Society Press, 439.448.

[9]. DERRICK, J. AND BOITEN, E. 1999. Testing refinements of state-based formal specifications. *Software Testing, Verification and Reliability 9*, 27.50.

[10]. FRANTZEN, L., TRETMANS, J., AND WILLEMSE, T. A. C. 2004. Test generation based on symbolic specifications. In *Formal Approaches to Software Testing, 4th International Workshop (FATES 2004)*. Linz, Austria, 1.15.

[11]. GERRARD, C. P., COLEMAN, D., AND GALLIMORE, R. M. 1990. Formal specification and design time testing. *IEEE Transactions on Software Engineering 16*, 12

[12]. GOGUEN, J. A. AND MALCOLM, G. 2000. *Software Engineering with OBJ: Algebraic Specifications in Action*. Kluwer Academic Publishers

[13]. LEE, D. AND YANNAKAKIS, M. 1996. Principles and methods of testing finite-state machines. *Proceedings of the IEEE 84*, 8, 1089.1123.

[14]. MACHADO, P. D. L. 2000. Testing from structured algebraic specifications. In *Algebraic Methodology and Software Technology (AMAST 2000)*. 529.544.

[15]. Bernot G., Testing against formal specifications: a theoretical view, TAPSOFT'91 CCPSD proceedings, LNCS n° 494, Springer-Verlag, Brighton, 1991, pp. 99-119

[16]. Desovski, D. Combining Testing and Model Checking for Verification of High Assurance Systems. IEEE Int. Symp. on High Assurance Software Engineering, IEEE (2004).

