2011 Fifth IEEE International Conference on Theoretical Aspects of Software Engineering

Automatic Specification-Based Testing: Challenges and Possibilities

Shaoying Liu
Department of Computer Science
Faculty of Computer and Information Sciences
Hosei University, Tokyo, Japan
Email: sliu@hosei.ac.jp

I. Introduction

Automatic specification-based testing (ASBT) has long been a goal of software engineering for its tremendous benefits in saving time and cost, but unfortunately due to many challenges and barriers, this goal still remains unrealized to a great extent. Specification-based testing (SBT) has been intensively researched under different names, such as “specification-based testing” [1], “black-box testing” or “functional testing” [2], or “model-based testing” [3]. The characteristic of SBT is that the test set is generated only based on the specification, without taking the internal structure of the program into account. ASBT is automated SBT, emphasizing that every activity in the testing process is done automatically. Examples of this technique include the *TestEra* tool reported by Khurshid and Marinov [4] and the Microsoft CHESS tool by Musuvathi *et al*. [5]. ASBT has two goals at different levels: *ideal goal* and *practical goal*. The ideal goal is to find all of the bugs in the program, which is well known to be extremely difficult, if not impossible. The practical goal is to meet required coverage criteria, such as statement coverage, branch coverage or path coverage. Although this may not ensure that all bugs are detected, it helps the developer gain confidence in the reliability of the program. Compared to the ideal goal, the practical goal is relatively easier to achieve but there are still many challenges lying ahead. In this paper, we discuss the major challenges, describe some possibilities in tackling the challenges, and present some open but interesting problems for future research in the field.

II. Challenges

Many issues concerned with SBT may be easy to deal with in manual testing but can become a challenge for ASBT. This includes specification language, methods for generating adequate test set, translation between abstract data and concrete data, analyzing test results to determine bugs, debugging, and integration testing.

A. Specification language

Conventional specification-based testing is usually performed manually because almost all of the specifications are written informally in practice. Using informal specification for automatic test set generation by a software tool requires that the algorithm of the tool automatically understands the informal language in which the specification is written, and extracts appropriate test conditions based on which test cases can be generated. However, due to the ambiguity of informal languages and the lack of well-defined structure of the informal specification, understanding the meaning of the specification has been a tremendous challenge, not need to mention automatic generation of test cases, as pointed out by Prasanna1 *et al.* [6]. Formal specification offers a much better basis for test case generation because the syntax and semantics of the specification are well defined, which facilitates the extraction of conditions suitable for test case generation and of test oracles for test result analysis [7], [8], [9]. However, this does not mean that all the potential problems have disappeared.

Further, when ASBT is applied in practice, a practical challenge is that it requires the availability of the formal specification. While writing a formal specification may effectively help requirements analysis, most practitioners find it difficult for real projects. Requiring the specification to accurately and completely reflect the user’s requirements is even more difficult since it needs to provide much details, which is likely to make the specification so complex that may not be easily understood, but without achieving the accuracy and completeness, the quality of the test set generated from the specification would be affected. How to handle this problem requires a practical solution.

Another practical challenge is that the module interface in the specification may not be consistent with that of the corresponding subprograms in the program, which is quite common in real projects. In this case, the gap between the module interface and that of the corresponding subprograms would be so great that prevents the seamless use of the test cases generated based on the module interface in testing those subprograms. How to address this problem poses another challenge for ASBT.

B. Methods for generating adequate test set

Adequate test set can be defined differently for different purposes. For example, a test set can be defined to be adequate if it allows to ensure the reliability of the program to a certain level; or alternatively it may require to meet certain program coverage criteria as mentioned before. In either case, it is

978-0-7695-4506-6/11 $26.00 © 2011 IEEE
DOI 10.1109/TASE.2011.36

extremely difficult to find a method that can theoretically establish a definitive relation between the test set generated using the method and the required adequacy standard, since there is a great variety of ways to implement the same specification into code. It seems that a feasible approach is to find out the capability of coverage of the test set, for example, through controlled experiments. Further, there is also a technical challenge when trying to produce a test case from a conjunction of predicates, such as $Q_1 \wedge Q_2 \wedge \cdots \wedge Q_n$ in which all the constituent predicates Q_i $(i = 1, ..., n)$ share the common input variables, say x_1, x_2, ..., x_m. If the data structure of the variables is complex, such as sequence of set of composite objects in which each field is again a nested data structure, the algorithm for generating test cases for these variables would be extremely hard to develop. Although we have put forward an approach of using model checking [10], [11], [12] to generate test cases from a conjunction of atomic predicates [13] and other researchers have developed some software tools such as TestEra [4] that uses the SAT-based Alloy [14] toolset to automatically find *instances* satisfying the formulae and the Yices SMT solver for the solution enumeration engine [15], their capacity seems currently limited to dealing with linear arithmetic and finite sets; not for all the types that are used in the popular formal notations. Furthermore, they do not address the issue whether the generated test cases satisfy any coverage or bug detection requirements.

C. Translation between abstract data and concrete data

Due to the different syntax of the specification notation and the programming language, abstract data used in the specification notation usually have a different format from that of concrete data in the program. Thus, test cases generated from the specification are usually not suitable for being directly used to run the program, because the compiler of the programming language usually does not recognize the format of the test cases. The test cases in the specification language format, which are called *abstract test cases*, must therefore be translated into the programming language format, which are called *concrete test cases*. A translator is needed for this purpose. Since an abstract variable may be implemented using different concrete data structures in different applications, the translator must be able to deal with all of the possibilities. For example, a sequence variable x may be implemented into an array, a vector, or a linked list. For sequence $x = [5, 9, 23, 35]$ in the specification notation, the corresponding programming language format is $x = \{5, 9, 23, 35\}$ (in Java). If x is implemented into a vector, a program segment must be created to form vector x containing the same elements. This challenge is an implementation challenge rather than a theoretical one, and relatively easy to handle than the challenge in producing adequate test set discussed previously.

D. Analyzing test results to determine bugs

A significant advantage of ASBT based on a formal specification in pre-post notation is that the test oracle, which is used for test result analysis to determine whether bugs are found by a test, can be precisely defined and automatically derived from the specification. let $S(S_{iv}, S_{ov})[S_{pre}, S_{post}]$ denote the specification of an operation S, where S_{iv} is the set of all input variables whose values are not changed by the operation, S_{ov} is the set of all output variables whose values are produced or updated by the operation, and S_{pre} and S_{post} are the pre- and post-conditions of S, respectively. A test oracle that can be derived from the specification is the following formula:

$S_{pre}(t) \wedge \neg S_{post}(t, P(t))$

where t denotes a test case and $P(t)$ the result of executing program P with t. When this formula evaluates to true on t, that is, pre-condition S_{pre} evaluates to true while post-condition S_{post} evaluates to false, a bug must be found by this test; otherwise, no bug is found this time.

Using such a well-defined formula for test result analysis requires the involvement of all of the output variables, which may include return value from an object's method call and state variables. The challenge is that the program may be implemented in the way that does not expose the values of some of the state variables as output of the program. Consequently, they cannot be used in evaluating the test oracle. Furthermore, the state variables may represent files or large-scale data structures, collecting them and letting them join the evaluation of the test oracle would consume a great deal of memory and time, which poses a challenge to the efficiency of ASBT tools.

E. Debugging

Debugging aims to locate bugs and determine their nature. Debugging is usually done by humans with or without a tool support, and completely automatic debugging still remains a challenging task. Of course, implementation-related bugs that can be defined without the need to refer to the specification are relatively easy to detect using static analysis tools [16], [17] or using a constraint-based approach [18]; the hardest part is to uncover semantic errors that can only be determined based on the specification or the user's requirements. The challenge is mainly attributed to the difficulty of establishing a definitive relation between the specification and all the possible states of program execution. All the states of a traversed program path can be collected (e.g., by using probes), but since specifications, in particular those in the form of pre- and post-conditions, usually focus only on the relation between the initial states and the final states (providing no information on the intermediate states), it is hard to know exactly which intermediate state starts to contain bugs (if any). To provide a solution for this problem, the key issue is whether the available information in the specification can help automatically examine every state of a traversed path and to judge whether the state contains any error.

F. Integration testing

ASBT for integration testing is more challenging than for unit testing. If the specification clearly shows how the operations are integrated using programming constructs such as explicit operation specifications in VDM, the information for operation integration can be found, but automatic extraction of the information may not be easy. The reason is similar to that for automatic structural testing of programs: difficult to handle loops. The specification notations such as the Structured Object-oriented Formal Language (SOFL) [19] that uses formalized data flow diagrams to describe system architectures have the similar challenge to the one mentioned above. One possibility to avoid this barrier is to write pre- and post-conditions for the operation to be tested, but if the operation is a high level operation that calls other operations, the post-condition may be too complex (if trying to cover the whole functionality) to be used or too simple (if only trying to cover partial functionality) to generate adequate test cases. In either circumstance, automatic test set generation would not be easy and satisfactory. Furthermore, one of the important tasks for integration testing is to test all of the useful functional scenarios. For example, in the case of an ATM software, it is important to test scenarios such as depositing sequentially twice with different amount and then withdrawing the total amount, but this functional scenario may not be directly defined in the specification. Test cases for exercising this kind of scenarios are almost impossible to automatically generate and so are their expected test results.

III. Possibilities

In this section, we present some strategies and approaches that can possibly deal with the most important challenges discussed previously.

A. Decompositional test case generation method

In our previous work [20], we put forward a decompositional test case generation method based on a pre-post style specification. let $S(S_{iv}, S_{ov})[S_{pre}, S_{post}]$ denote the specification of an operation S. The essential idea is first to transform such a specification into an equivalent disjunction of *functional scenarios* $({}^{\sim}S_{pre} \wedge C_1 \wedge D_1) \vee ({}^{\sim}S_{pre} \wedge C_2 \wedge D_2) \vee \cdots \vee ({}^{\sim}S_{pre} \wedge C_n \wedge D_n)$ and then generate test sets and analyze test results based on the functional scenarios. A functional scenario ${}^{\sim}S_{pre} \wedge C_i \wedge D_i$ $(i = 1...n)$ is a conjunction of the decorated pre-condition ${}^{\sim}S_{pre}$, *guard condition* C_i (containing only input variables), and *defining condition* D_i (containing at least one output variable); it tells clearly that the output is defined by defining condition D_i when the input satisfies the *test condition* ${}^{\sim}S_{pre} \wedge C_i$.

Generating a test set based on the specification is realized by generating it from its all functional scenarios. The production of a test set from a functional scenario is done by generating it from its test condition, which can be divided further into test case generations from every disjunctive clause of the test condition. We use $\mathcal{G}$ to denote a *test set generator* (TSG), which is a function from the universal set of logical expressions to the universal set of test sets. Thus, $\mathcal{G}(p)$ represents the test set generated to satisfy predicate p (i.e., every test case in the set satisfies p). Below are the major criteria for generating a test set from the specification using the decompositional method.

Criterion 1: $\mathcal{G}(({}^{\sim}S_{pre} \wedge C_1 \wedge D_1) \vee ({}^{\sim}S_{pre} \wedge C_2 \wedge D_2) \vee \cdots \vee ({}^{\sim}S_{pre} \wedge C_n \wedge D_n)) = \mathcal{G}({}^{\sim}S_{pre} \wedge C_1 \wedge D_1) \cup \mathcal{G}({}^{\sim}S_{pre} \wedge C_2 \wedge D_2) \cup \cdots \cup \mathcal{G}({}^{\sim}S_{pre} \wedge C_n \wedge D_n)$.

Since the execution of a program only requires input values, test case generation from a functional scenario actually depends only on its test condition, which is reflected in **Criterion 2** next.

Criterion 2: $\mathcal{G}({}^{\sim}S_{pre} \wedge C_i \wedge D_i) = \mathcal{G}({}^{\sim}S_{pre} \wedge C_i)$

In general, pre-condition ${}^{\sim}S_{pre}$ can be in any form of predicate expression. To define $\mathcal{G}$ to deal with test case generation from the conjunction ${}^{\sim}S_{pre} \wedge C_i$, a systematic way is first to translate the conjunction into an equivalent disjunctive normal form (DNF) and then define $\mathcal{G}$ based on it, which is formally expressed by **Criteria** 3 and 4.

Criterion 3: Let $P_1 \vee P_2 \vee \cdots \vee P_m$ be a DNF of the test condition ${}^{\sim}S_{pre} \wedge C_i$. Then, we define $\mathcal{G}({}^{\sim}S_{pre} \wedge C_i) = \mathcal{G}(P_1 \vee P_2 \vee \cdots \vee P_m) = \mathcal{G}(P_1) \cup \mathcal{G}(P_2) \cup \cdots \cup \mathcal{G}(P_m)$.

Each P_i $(i = 1, ..., m)$ is a conjunction of atomic predicates, say $Q_i^1 \wedge Q_i^2 \wedge \cdots \wedge Q_i^w$. An atomic predicate expression, in our context, is one of the two kinds of predicates: (1) a relation (e.g., $x > y$) and (2) a negation of a relation.

Criterion 4: $\mathcal{G}(Q_i^1 \wedge Q_i^2 \wedge \cdots \wedge Q_i^w) = \mathcal{G}(Q_i^1) \cap \mathcal{G}(Q_i^2) \cap$, ..., $\cap\ \mathcal{G}(Q_i^w)$

The remaining problem in test set generation is how to generate a test set for a single atomic predicate; that is, to define, for example, $\mathcal{G}(Q_i^1)$. **Criterion** 5 deals with this issue.

Criterion 5: Let $S_{iv} = \{x_1, x_2, ..., x_r\}$ and $Q(x_1, x_2, ..., x_q)$ $(q \leq r)$ be a relation involving variables $x_1, x_2, ..., x_q$. Then, $\mathcal{G}(Q(x_1, x_2, ..., x_q)) \subseteq \{T_c \mid (\forall_{x \in \{x_1, x_2, ..., x_q\}} \cdot Q(T_c(x_1), T_c(x_2), ..., T_c(x_q))) \wedge (\forall_{x \in (S_{iv} \setminus \{x_1, x_2, ..., x_q\})} \cdot T_c(x) = any)\}$

where T_c denotes a test case, which is a set of pairs of an input variable and its value (e.g., $T_c = ((x_1, 2), (x_2, 6),$ $..., (x_r, 9))$, $T_c(x_i)$ $(i = 1, 2, ...r)$ represents the value of x_i in test case T_c, and $T_c(x) = any$ means any value for variable x, taken from its type randomly. For example, assume that Q_i^1 is $x > 0$ and Q_i^2 is $x < y$, and all of the input variables are x, y, and z. We can then create one test case for each predicate: $t_1 = \{(x, 2), (y, 0), (z, 8)\}$ satisfying Q_i^1 and $t_2 = \{(x, 2), (y, 5), (z, 8)\}$ satisfying Q_i^2. If Q_i^1 and Q_i^2 are expected to be the same, the same variable in both test cases must be assigned the same value. Note that the definition of the test set generated for an atomic predicate is nondeterministic, allowing the tool builder to freely choose appropriate algorithm to implement it.

B. Automatic Debugging

We present the preliminary idea of a new debugging technique in this section. The underlying idea comes from a combination of the functional scenario-based testing discussed previously and Floyd-Hoare logic for proving program correctness [21]. Consider the case of using a single test case t, which is generated from the test condition ${}^{\sim}S_{pre} \wedge C_i$ of the functional scenario ${}^{\sim}S_{pre} \wedge C_i \wedge D_i$, to execute the program P under test. To facilitate the illustration, we assume that the input variables of the operation are x, y, and z, and output variables are u, v, and w; that is, $\tilde{}S_{pre} \wedge C_i$ only involves x, y, and z, and D_i contains u, v, and w as well as some or all of the input variables. When executing program P with test case t, we will be able to get a traversed program path, say $p = [c_1, c_2, ..., c_m]$ where each c_i $(i = 1, 2, ..., m)$ is either an assignment statement or condition of the program. At this point, we already know that a bug exists in this path, but where exactly the bug is located is unknown. To determine its precise location, we form the "Hoare triple" (a specialized version in the sense that p is not the entire program but a single program path, and $C_i \wedge D_i$ is not the entire post-condition but the most relevant part of the post-condition):

$$\{\tilde{}S_{pre}\}\ p\ \{C_i \wedge D_i\}\ .$$

If path p is found to have a bug by test case t with respect to the pre-condition $\{\tilde{}S_{pre}\}$ and the "post-condition" $\{C_i \wedge D_i\}$, by definition, $\tilde{}S_{pre}(t)$ must evaluate to true and $C_i \wedge D_i(t, r)$ evaluates to false, where r denotes the result of program execution based on t. In order to locate the bug on the path, we need to properly rename input variables to distinguish them from intermediate state variables and then repeatedly apply the axiom of assignment in the Floyd-Hoare logic starting from the "post-condition" until the bug is found. All the steps can be automatically supported. There are still many challenging issues to be addressed in this approach, especially when complex data structures are involved. For example, there is still a lack of a mechanism for *accurately* determining the location of bugs in a program path. Completely addressing these problems requires further research.

IV. Conclusion and Future Work

This paper discusses the major challenges for automatic specification-based testing. We describe a decompositional method for test case generation based on a pre-post style specification and the preliminary idea of a new debugging technique as a potential solution for the challenging problems. Although there is much work remaining to be done in this direction, our discussion aims to set up a basis for further developments.

There are several interesting problems to be studied in the future. The first is to establish a theory that explains the relation among specification, test case generation method, coverage criteria, and bug detection effectiveness. The goal of this research is to understand what specification can be used by what method to generate an adequate test set that meets the required coverage criteria and/or bug detection effectiveness. The second is to build a set of specific inter-related "vibration" techniques (or the ways of vibration) for generating adequate test sets from test conditions to meet various coverage criteria. The third is to further develop the automatic debugging idea presented in this paper into a mature technique. The final problem to study is how to build software tools that can effectively and efficiently support the whole process of automatic specification-based testing and bug removal.

V. Acknowledgement

This work is supported by NII Collaborative Research Program. It is also supported by the NSFC Grant (No. 60910004) and 973 Program of China Grant (No. 2010CB328102).

References

[1] J. Goodenough and S. Gerhart. Toward a Theory of Test Data Selection. *IEEE Transactions on Software Engineering*, 1(2):156–173, June 1975.

[2] B. Beizer. *Black-Box Testing*. John Wiley and Sons, Inc., 1995.

[3] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-Based Testing in Practice. In *Proceedings of Proceedings of the 21st International Conference on Software Engineering*, pages 285–294. IEEE Computer Society Press, 1999.

[4] S. Khurshid and D. Marinov. TestEra: Specification-based Testing of Java Programs using SAT. *Automated Software Engineering*, 11(4), 2004.

[5] M. Musuvathi, S. Qadeer, and T. Ball. CHESS: A Systematic Testing Tool for Concurrent Software. Technical Report MSR-TR-2007-149, Microsoft Research, 2007.

[6] M. Prasanna1, S. N. Sivanandam, R. Venkatesan, and R. Sundarrajan. A Survey on Automatic Test Case Generation. *Academic Open Internet Journal*, 15, 2005.

[7] I. J. Hayes. Specification Directed Module Testing. *IEEE Transactions on Software Engineering*, SE-12(1):124–133, January 1986.

[8] J. Dick and A. Faivre. Automating the Generation and Sequencing of Test Cases from Model-based Specifications. In *Proceedings of FME '93: Industrial-Strength Formal Methods*, pages 268–284, Odense, Denmark, 1993. Springer-Verlag Lecture Notes in Computer Science Volume 670.

[9] P. Stocks and D. Carrington. Test Template Framework: A specification-based testing case study. In *ISSTA93*, pages 11–18, Cambridge MA, June 1993.

[10] E. M. Clarke, O. Grumber, and D. Peled. *Model Checking*. MIT Press, 2000.

[11] Z. Duan, C. Tian, and L. Zhang. A decision procedure for propositional projection temporal logic with infinite models. *Acta Informatica*, 45(1):43–78, Feb. 2008.

[12] Z. Duan and C. Tian. A Unified Model Checking Approach with Projection Temporal Logic. In *Proceedings of 2008 International Conference on Formal Engineering Methods (ICFEM 2008)*, pages 167–186, Kitakyushu, Japan, Oct. 2008. LNCS 5256, Springer.

[13] C. Tian, S. Liu, and S. Nakajima. Utilizing Model Checking for Automatic Test Case Generation from Conjunctions of Atomic Predicate Expressions. In *Proceedings of 3rd International Workshop on Constraints in Software Testing, Verification, and Analysis*, Berlin, Germany, March 2011. IEEE CS Press.

[14] Daniel Jackson. Alloy: a lightweight object Modelling notation. Technical Report 797, MIT Laboratory for Computer Science, Cambridge, MA, February 2000.

[15] B. Dutertre and L. de Moura. The YICES SMT Solver. Technical report, SRI International, USA.

[16] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. Mcpeak, and D. Engler. A Few Billion Lines of Cod Later Using Static Analysis to Find Bugs in the Real World. *Communications of the ACM*, 53(2):66–75, Feb. 2010.

[17] http://www.coverity.comStatic Analysis Tool. *Converity Static Analysis*.

[18] F. Wotawa, J. Weber, M. Nica, and R. Ceballos. On the Complexity of Program Debugging using Constraints for Modelling the Program's Syntax and Semantics. In *Proceedings of 13th Conference on Current Topics in Artificial Intelligence (CAEPIA 2009)*, pages 22–31, Seville, Spain, November 9-13 2009. LNCS, Springer-Verlag.

[19] S. Liu. *Formal Engineering for Industrial Software Development Using the SOFL Method*. Springer-Verlag, ISBN 3-540-20602-7, 2004.

[20] S. Liu and S. Nakajima. A Decompositional Approach to Automatic Test Case Generation Based on Formal Specifications. In *4th IEEE International Conference on Secure Software Integration and Reliability Improvement (SSIRI 2010)*, pages 147–155, Singapore, June 9-11 2010. IEEE CS Press.

[21] C.A.R. Hoare and C. B. Jones. *Essays in Computer Science*. Prentice Hall International, 1989.