International Journal of Software Engineering and Knowledge Engineering
Vol. 10, No. 2 (2000) 00–00

TEST SCENARIO GENENERATION BASED ON FORMAL SPECIFICATION AND USAGE PROFILE

KAI H. CHANG, SHIH-SUNG LIAO, RICHARD CHAPMAN and CHUN-YU CHEN

Department of Computer Science and Engineering,
Auburn University, Auburn, AL 36849, USA
E-mail: {*kchang,ssliao,chapman,chunchen*}*@eng.auburn.edu*

Received 21 August 1997
First Revision 17 December 1997
Second Revision 7 August 1998
Accepted 18 August 1998

This paper presents a method for test scenario generation based on formal specifications and usage profiles. It is a major component of a framework for testing object-oriented programs. In this framework, the requirements of a software system are formally specified. The anticipated application of the system is expressed in a usage profile, which is a state model that indicates the dynamic behavior of the system and execution probabilities for the behaviors. The state model is used as a guide to derive the anticipated operation scenarios. An enhanced state transition diagram is used to represent the state model, which incorporates hierarchy, usage and parameter information. Since the number of feasible scenarios can be extremely large, probability and importance criteria are used to select the most probable and important scenarios.

Keywords: Formal methods; software testing; Object-Z; software validation and verification.

1. Introduction

Although numerous testing adequacy criteria have been investigated or experimented, none of them can be proven to be better than others for an arbitrary program [7]. Musa, a major advocate for *usage-based testing* (UBT), insists that testing should be based on the usage information [11]. From the perspective of testing, the most-used operations should be tested more thoroughly to ensure their reliability. In terms of *mean time between failures*, usage testing improves the software reliability by a factor of 21 compared to using coverage testing based on the reports from a number of projects [4]. Both theoretical and practical experiences indicate that the usage profile can provide a great means for improving software quality from the user's perspective. The statistical testing of the *Cleanroom* development [4, 6] approach is also a UBT method. A stochastic grammar is used to describe the usage distribution in Cleanroom [10]. An approach that applies usage information to distribute testing efforts among modules based on their struc-

ture dependency is given in [19]. These approaches have all been used for testing structured programs.

Specification-based testing (SBT) is a technique that incorporates the specification into the testing process. The rationale for the SBT technique is that all software development should begin with a requirement specification. Stocks and Carrington present a specification-based testing framework that is designed for any model-based specification notation [17]. Richardson *et al.* [13] combine real time interval logic and the Z specification language [16] to derive test oracles from specifications and incorporate them in the testing process for reactive systems. A summary of some major efforts that apply formal specification languages to software testing can be found in [2]. It is noteworthy that none of these SBT approaches uses an object-oriented specification language.

This paper is directed toward testing object-oriented software. However, this technique can be applied to conventional software as well. The main features include:

- Use of object-oriented formal specifications to speed up the prototyping and implementation of OO programs, and ultimately provide a basis for testing.
- Use of state model to specify the dynamic behavior of the system in terms of the anticipated operation sequences and usage probabilities.

While the overall approach will be discussed, this paper will focus on the test scenario generation portion of the approach. The next section outlines the overall approach. Section 3 introduces the enhanced state transition diagram. Section 4 discusses the derivation of test scenarios. Section 5 gives an illustrative example, and Sec. 6 concludes the paper.

2. The Framework

Although the underlying ideas of this approach are language independent, the implementation language is an important factor for an effective object-oriented design. This framework is targeted at testing programs written in C++. The formal specification language adopted in this framework is Object-Z [5]. Object-Z [1] is an extension of Z which embraces the object-oriented style, especially in the aspects of object, class, inheritance and composition, while still keeping the advantages of Z@. It provides a mechanism for defining class structure and gives a precise description of the desired operational behavior. It has been noted that there is a mapping between the OO specification and C++ [12]. In addition to helping implementation, i.e., the transformation from specification to program, the mapping can also serve as a guide for testing and verifying test results.

The dynamic behavior of a system (i.e., the operation sequence of a system) may not be completely defined in the Object-Z specifications. In this paper, it would be described in a state transition diagram (STD). When the STD is associated with probability information, it can be used as an operation profile to model the usage of

the system. The operation usage profile can be expressed in a formal model, called *enhanced state transition diagrams* (ESTDs) [9, 3]. At the highest level, an ESTD is constructed according to the major operation steps. At the lowest level, a state must be associated with an operation in a class.

Test scenarios are derived from the operation usage profile. A test scenario is defined as a sequence of operations; it can be represented by a sequence of events or states. The probability of a test scenario indicates the frequency of its usage. After a test scenario is constructed, test cases can be generated according to the data profile. A data profile specified in an *extended Backus-Naur Form* (EBNF) [9] describes the conventional data type definition as well as its distribution. Any number of test cases can then be generated according to the specified data distribution.

3. Enhanced State Transition Diagram

An enhanced state transition diagram (ESTD) [3, 9] is used to represent the state model of a system. It is derived from the object-oriented specification and the usage description of the system. In this model, hierarchy, usage, and parameter information are included. The hierarchical feature is achieved by using a '*' next to a state to indicate that the state can be further refined. This allows hierarchical organization of the specification and provides a clear mapping between the specification and the state model. The hierarchical nature also helps to control the state explosion problem of complex systems. The probability associated with each transition indicates its usage frequency. Information about input and output parameters, functions that are called, and pre- and postconditions is also provided in the diagram. The names of the input and output parameters will be used by the Test Data Generator to generate input data and by the Execution Monitor to check output results. The pre- and postconditions are obtained from the Object-Z specification and can be interpreted by the tester, or eventually by an oracle tool. Detailed ESTD examples will be given in a later section.

3.1. *Usage profile*

A usage profile is used to indicate the relative execution frequencies of various branchings of a software system. Its purpose is not to obtain precise execution probabilities, but to establish the relative weights that allows testing efforts to be distributed according to the usage. In addition to usage, a profile can be purposely modified to reflect the importance of certain branchings. This would guarantee that rarely executed, but critical paths will receive adequate testing efforts. It should also be noted that the usage of a software can be modeled by several profiles. The selection may depend on the users group, environment, and emphasis of testing. There are several approaches that a usage profile can be obtained [19].

If a similar system already exists, its usage data can be used as a reference. If a new system to be developed is to automate existing manual activities, the manual usage information can also be gathered and analyzed. If the system to be

implemented is a unique or brand new application, a preliminary state transition diagram should be constructed based on design or specification. Two methods can be used to assign the initial usage profile: (1) uniform weight is assigned, and (2) non-uniform weights are assigned based on the users' intuition. A tool can then be used to adjust the weights interactively while maintaining the sum of probabilities of each branching point to be one. Several usage profiles may be designed for different applications or different user groups.

3.2. *Data flow analysis for ESTD derivation*

Automatic or machine-aided derivation of the ESTD from an Object-Z specification is an area of current work. Three possible methods are being investigated to address generation of the ESTD.

In the simplest approach, we require the specification writer to explicitly identify a variable at each level of the hierarchy whose value denotes the state of the system. Such state variables can only take values from enumerated types. The ATM example included later in this paper is an example of such a specification. However, there are obvious drawbacks to such a scheme — the specification writer is constrained to write particular sorts of specifications that may or may not mesh with the sort of specification that would be best.

Of course, to automatically generate an ESTD from an arbitrary specification is an undecidable task, so some limits on the specification must be accepted as inevitable. Thus, our approach to automatic ESTD generation is to attempt to derive the sequence of state transititons at each level through dataflow-style analysis of the schemata in the specification. Consider the following Object-Z specification for a stack.

Defining a state as an assignment of values to variables, the state diagram might look like Fig. 1(a). However, such a diagram is probably not what we would like to build; it captures too much detail about the system, and in particular, the state diagram we draw depends on how the type T is instantiated. If we were to abstract away from the actual contents of the state variable *items*, and use only *#items* to denote the state, we get the state diagram of Fig. 1(b).

A third approach would be building states based on the truth-values of propositions and predicates mentioned in the specification. Thus a state would correspond to an assignment of truth-values to each proposition from the relevant set. An example of such a resulting state diagram might be the 3-state diagram of Fig. 1(c). Only three states are depicted because it is not possible for the number of items to be zero and greater than *max*. However, detecting that fact is probably beyond the capability of an automated ESTD-construction system.

Using compiler-style dataflow analysis on the schema text, we can compute how state variables and output variables of schema are modified by a valid implementation. The task is somewhat more difficult than would be the case for an imperative programming language, since the predicates used in a schema are not ordered, but

Stack[T]

$max : \mathbb{Z}$

$max = 100$

$items : \text{seq } T$

$\#items \leq max$

INIT

$items = \langle\rangle$

push

$\Delta(items)$
$item? : T$

$\#items < max$
$items' = \langle item? \rangle \frown items$

pop

$\Delta(items)$
$item! : T$

$items \neq \langle\rangle$
$items' = \langle item! \rangle \frown items'$

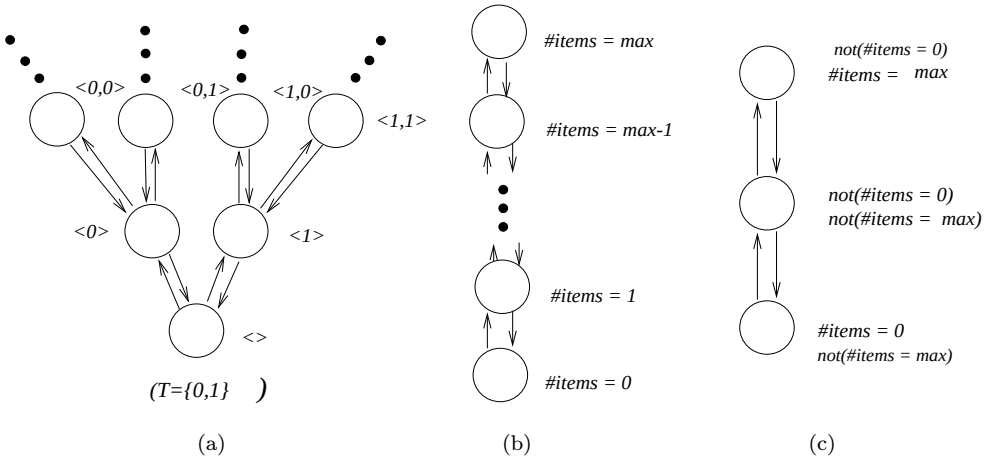

Fig. 1. Three ESTD's for a stack.

we can still derive directed graphs that relate inputs and initial values of state variables in a schema to the outputs and final values of the state variables, perhaps via a sequence of intermediate operations.

The (partial-order) relations that exist between the states (which are the pre- and postconditions of the operations or function calls specified in a schema) are used to build the state-transition diagram. For example, in the stack specification above,

Table 8. Sample test scenario for Path 1 of Table 7.

Test Scenario: $v_1v_2v_3v_4v_{5.1}v_{5.2}v_{5.1}v_{5.6}v_{5.7}v_{5.8}v_{5.9}v_7$ Weight: 0.0504			
Current state	Function list	Next State	Event
1	{ }	2	card inserted
2	ATMProcess.CardVerify(+,-)	3	valid card
3	ATMProcess.PinVerify(+,-)	4	valid PIN entered
4	ATMProcess.GetAccount(+,-)	5.1	account information received
5.1	ATMProcess.ProcessTrans(+)	5.2	transaction selection
5.2	ATMProcess.Withdraw(+,-)	5.1	withdraw transaction
5.1	ATMProcess.ProcessTrans(+)	5.6	transaction selection
5.6	ATMProcess.EndTransaction(+,-)	5.7	closing transaction
5.7	ATMProcess.UnLock(+,-)	5.8	accounts unlocked
5.8	CardReader.EjectCard(+,-)	5.9	card ejected
5.9	ATMProcess.ProcessTrans(-)	7	transaction completed
7	{ }	–	

incorporated. In this case, the exploration of the state hierarchy and the execution can be visualized in the form of state traversal. It is expected that, with the use of usage-based testing, the quality of software can be greatly improved.

References

1. D. Carrington, R. Duke, P. King, G. Rose, and G. Smith, "Object-Z: an object-oriented extension to Z", in *Proc. IFIP TC/WG 6.1 Second Int. Conf. on Formal Description Techniques for Distributed Systems and Communications Protocols*, Vancouver, Canada, December 1989, pp. 281–296.
2. D. Carrington and P. Stocks, "A tale of two paradigms: formal methods and software testing", in *Workshops in Computing Series: Z User Workshop*, Springer-Verlag, 1994, pp. 51–68.
3. K. H. Chang, S. S. Liao, S. B. Seidman, and R. Chapman, "Testing Object-Oriented Program: From Formal Specification to Test Scenario Generation", *Journal of Systems and Software*, to appear in 1998.
4. R. H. Cobb and H. D. Mills, "Engineering software under statistical quality control", *IEEE Software*, November 1990, pp. 44–54.
5. R. Duke, P. King, G. Rose, and G. Smith, "The Object-Z specification language, Version 1", Technical Report 91-1, Software Verification Research Centre, Department of Computer Science, University of Queensland, May 1991.
6. M. Dyer, *The Cleanroom Approach to Quality Software Development*, John Wiley, 1992.
7. P. G. Frankl and S. N. Weiss, "An experimental comparison of the effectiveness of branch testing and data flow testing". *IEEE Trans. on Soft. Eng.* **19**, (1993) 774–787.
8. W. Johnston, "A type checker for object-z", Technical report, Software Verification Research Centre, University of Queesnland, 1996.
9. S. Liao, *An Integrated Testing Approach for Object-Oriented Programs*, PhD Dissertation, Department of Computer Science and Engineering, Auburn University, March 1997.

10. R. C. Linger and H. D. Mills, "A case study in Cleanroom software engineering: the IBM Cobol restructuring facility", in *Proc. COMPSAC'88*, 1988.
11. J. D. Musa, "Operational profiles in software reliability engineering". *IEEE Software*, March 1994, pp. 14–32.
12. G.-H. B. Rafsanjani and S. J. Colwill, "From Object-Z to C++: A Structure Mapping", in *Workshops in Computing Series: Z User Workshop*, Springer-Verlag, Dec. 1992, pp. 166–179.
13. D. J. Richardson, S. L. Aha, and T. O. O'Malley, "Specification-based test oracles for reactive systems", in *Proc. 14th Int. Conf. on Software Engineering*, May 1992, pp. 105–118.
14. R. Sedgewick, *Algorithms*, Addison-Wesley, 1988.
15. J. M. Spivey, *Understanding Z*, Cambridge University Press, 1988.
16. J. M. Spivey, *The Z notation: a reference manual, 2nd ed.*, Prentice-Hall, 1992.
17. P. A. Stocks and D. A. Carrington, "Test templates: a specification-based testing framework", in *Proc. 15th Int. Conf. on Software Engineering*, pages 405–414, Los Alamitos, CA, 1993.
18. J. A. Whittaker, "Markov analysis of software specification", *ACM Trans. Soft. Eng. Methodology* **2** (1993) 93–106.
19. C. Wohlin and P. Runeson, "Certification of software components", *IEEE Trans. Software. Eng.* **20** (1994) 494–499.