Generating Class-Based Test Cases for Interface Classes of Object-Oriented Black Box Frameworks

Jehad Al Dallal, and Paul Sorenson

International Science Index, Computer and Systems Engineering Vol:2, No:4, 2008 waset.org/Publication/12038

***Abstract*—**An application framework provides a reusable design and implementation for a family of software systems. Application developers extend the framework to build their particular applications using hooks. Hooks are the places identified to show how to use and customize the framework. Hooks define the Framework Interface Classes (FICs) and their possible specifications, which helps in building reusable test cases for the implementations of these classes. This paper introduces a novel technique called all paths-state to generate state-based test cases to test the FICs at class level. The technique is experimentally evaluated. The empirical evaluation shows that all paths-state technique produces test cases with a high degree of coverage for the specifications of the implemented FICs comparing to test cases generated using round-trip path and all-transition techniques.

***Keywords*—**Hooks, object-oriented framework, framework interface classes (FICs), specification-based testing, test case generation.

I. INTRODUCTION

AN application framework provides a reusable design and implementation for a family of software systems [1]. It contains a collection of reusable concrete and abstract classes. The framework design provides the context in which the classes are used. The framework itself is not complete. Users of the framework complete or extend the framework to build their particular applications. Places at which users can add their own classes are called *hooks* [2,3,4]. Frameworks are classified according to their customization method into two categories [5]: white box and black box. In white box frameworks, the functionality is extended or customized by subclassing some existing framework classes. In the black box frameworks, compositions and existing components are used without inheritance. Gray-box frameworks contain the characteristics of both black and white-box frameworks.

Jehad Al Dallal is with Department of Information Sciences, Kuwait University, P.O. Box 5969, Safat 13060, Kuwait (e-mail: jehad@cfw.kuniv.edu).
Paul Sorenson is with Department of Computing Science, University of Alberta, Edmonton, AB. T6G 2H1, Canada (e-mail: sorenson@cs.ualberta.ca).

To build an application using a framework, application developers create two types of classes: (1) classes that use the framework classes and (2) classes that do not. Classes that use the framework classes are called Framework Interface Classes (FICs) because they act as interfaces between the framework classes and the second type of the classes created by application developers. Instances of FICs are called framework interface objects. Fig. 1 shows the relation between the framework classes, the hooks, and the FICs. FICs use the framework classes in two ways: either by sub-classing them (i.e., when using white box frameworks) or by using them without inheritance (i.e., when using black box frameworks). In this paper, the focus is in testing FICs that use framework classes without inheritance. Hooks define how to use the framework, and therefore, they define the FICs and specify the pre-conditions and post-conditions of the FIC methods. Synthesizing the FIC state-based model from the pre-conditions and post-conditions of the FIC methods is detailed in [6].

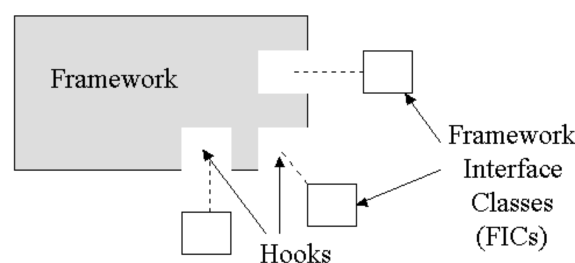

Fig. 1 Framework interface classes

Application developers may use all FICs or some of them according to the application requirements. When application developers use FICs to implement their applications, they deal with the specification of the FICs introduced by the hooks in three ways: (1) use them as defined, (2) add new specifications for the added behaviors to meet the application requirements, and (3) ignore specifications for the behaviors that are unnecessary in implementing the application requirements. The FIC specifications can be represented using a State Transition Diagram (STD) or an UML statechart.

Fig. 2 shows the STD representation of a *NewAccount* banking framework interface object specification introduced by the framework hooks. The STD contains two special states: *alpha* and *omega* to represent the states of the object before

being constructed and after being destructed. Moreover, the STD contains the *Open, Overdrawn, Inactive,* and *Frozen* states to model the states of the object. A banking system developer may choose to implement the specification shown in Fig 2 as defined. Moreover, the developer may choose to add, for example, transitions between *Overdrawn* and *Frozen* states to match the requirement of a banking system. Finally, the application developer may choose to ignore, for example, the transition originated from the *Open* state and ended at the *Inactive* state. This implies that an account can never go directly from the *Open* state to the *Inactive* state without going though the *Frozen* state first.

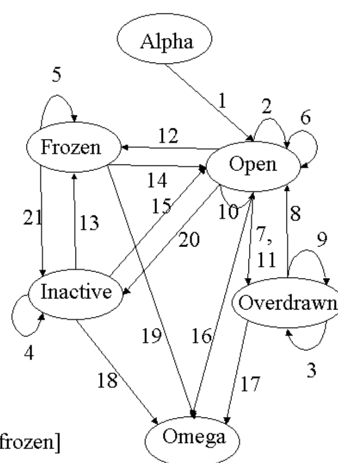

Fig. 2 The STD of the *NewAccount* object defined in the banking framework hooks

Testing techniques for deriving test cases starting from the software specification only are called specification-based testing techniques. Instead of applying such techniques to produce test cases to test the FICs every time a framework application is developed, we can apply them once to build reusable test cases when the framework is developed and the hooks that specify the FICs behaviors are described. The test cases that are generated at this stage are called *baseline test cases*. When developing the framework application the developer can reuse some of the baseline test cases and write new test cases only for the added behaviors instead of writing all test cases from scratch.

As a result, two main problems have to be tackled: (1) building effective baseline test cases in terms of reusability and fault coverage, and (2) introducing an efficient way to use the baseline test cases. Although we are studying both problems, this paper focuses on just the first problem. The drawbacks of using the existing testing techniques are studied and a novel test case generator technique that eliminates the drawbacks is introduced. Finally, an empirical study is reported to show the relative effectiveness of the test cases generated using the proposed technique in covering the specifications of the implemented FICs.

The paper is organized as follows. Section II discusses the related work. In Section III, the proposed test case generator technique, *all paths-state*, is described. An empirical evaluation is reported in Section IV. Finally, Section V provides conclusion and discussion of future work.

II. Related Work

In object-oriented testing, each class has to be tested individually. Class testing is a unit testing step with respect to application testing and the first level of integration testing. At class testing level, the method responsibilities, intraclass interactions, and superclass/subclass interactions are considered [7]. Research in generating test cases to test an implementation at the class level can be divided into two broad approaches: (1) generating test cases from the source code to achieve a given level of statement, branch, or path coverage, and (2) generating test cases from the formal specification of the implementation. Testing techniques that follow the former approach are called implementation-based testing techniques (also sometimes referred to as white-box testing techniques), while testing techniques that follow the latter approach are called specification-based testing techniques (also sometimes referred to as black-box testing techniques).

The specification of a class behavior can be expressed using state-based models such as finite state machines and UML statecharts [7]. In this case, a state is a set of instance variable value combinations of the class object. A transition is an allowable two-state sequence caused by an event. An event is a method call. Each transition may be associated with (1) an event, (2) a set of predicates, and (3) a set of expected actions. The UML syntax for a transition is:

event-name argument-list [*guard predicate*]/*action-expression*

There are several state-based specification coverage criteria proposed in the literature such as:

1. All-transition coverage. In all-transitions coverage, each transition is covered at least once in some test case. Therefore, to test a transition, the test case requires that the object under test be in the accepting state of the transition. The technique does not put any constraints on how to reach the accepting state. Chow [8] introduced the all-transition coverage technique for finite state machines and Offut et al. [9] adapted the technique for UML statecharts and compared it experimentally with other specification coverage techniques. Bogdanov et al. [10] used the all-transitions coverage technique to derive test sequences in the presence of hierarchical statecharts.
2. Transition-pair coverage. In transition-pair coverage, it is required to cover each pair of adjacent transitions at least once in some test case [8,9,11]. Therefore, the transition-pair coverage subsumes the all-transitions coverage.
3. Full predicate coverage. In full predicate coverage, it is required to cover each clause in each predicate on every transition, if the clause independently affects the value of the predicate [9,11]. Offut et al. [9] showed experimentally that the full predicate coverage is more effective than the transition-pair coverage in terms of fault coverage. Abdurazik et al. [11] compared experimentally the transition-pair coverage and the full predicate coverage. The comparison results showed that transition-pair tests offer something different from full predicate tests. Moreover, the comparison

International Science Index, Computer and Systems Engineering Vol:2, No:4, 2008 waset.org/Publication/12038

results showed that the test set size of the transition-pair coverage technique is larger than the test set size of the full predicate coverage technique. This means that applying the transition-pair coverage technique costs more than applying the full predicate coverage technique.

4. Round-trip path coverage. In round-trip path coverage, transition sequences that start and end with the same state and simple paths from *alpha* to *omega* state are covered. A simple path includes only an iteration of a loop, if a loop exists in some sequence. The round-trip path coverage guarantees that each transition in the model is covered at least once and, therefore, it subsumes the all-transitions coverage. The round-trip path strategy was proposed originally by Chow [8] and was denoted as W-method. Binder [7] adapted the strategy to UML statecharts and called it round-trip path testing. Antoniol et al. [12] showed experimentally that the round-trip path testing strategy is reasonably effective at detecting faults. Kim et al. [13] used a technique similar to the round-trip path strategy to derive testing trees for testing control and data flow through states.

FICs are not framework classes. They are not implemented unless an application developer uses the framework hooks to implement them at the application development stage. However, since the framework hooks introduce the specifications of the FICs, the test cases that can be used to test the FICs at the application testing stage can be produced once when the hooks are described and applied each time the FICs are used to develop an application. When any of the existing coverage techniques, except the transition-pair coverage, is applied to generate baseline test cases for FICs, only one transition sequence is required to cover a transition. For the example STD given in Fig. 2, in the *all-transitions* technique, to cover the transition labeled 15, we can follow the path that has the sequence of transitions (1,20,15) and we do not have to worry about any other paths such as (1,12,21,15). In the transition-pair coverage, some but not all transition sequences are used to cover a transition. For example, to cover the transition-pair (15,7), we can follow the path that has the sequence of transitions (1,20,15,7) and we do not have to worry about any other paths such as (1,12,21,15,7). The sequences of transitions are used to derive the required test cases.

The application developer can decide to ignore some of the specifications for the FIC behaviors because they are unnecessary in implementing the application. Therefore, any baseline test case derived from a sequence of transitions that includes an unimplemented transition is considered broken and cannot be used as-is. Consequently, the application tester has to build new test cases or modify some baseline test cases to test the implemented transitions that were supposed to be tested using the broken baseline test cases.

For example, when the round-trip path strategy is used to derive test cases for the NewAccount FIC (the STD is shown in Fig. 2), the tree shown in Fig. 3 is constructed. Each path from the root node to a leaf node is used to build a test case. Since there are 16 such paths, 16 test cases are built. If the application developer chooses not to implement the transition originating from the Open state and ending at the Inactive state, the test cases built using the round-trip paths that include the transition are considered broken, and therefore, they cannot be used as-is. This results in breaking the test cases built from the paths that include the transition sequences labeled as (1,20,13), (1,20,15), (1,20,18), and (1,20,4). Note that the outgoing transitions from the Inactive state may be implemented in the application, but none of the non-broken test cases, built using the round-trip path strategy, can test them.

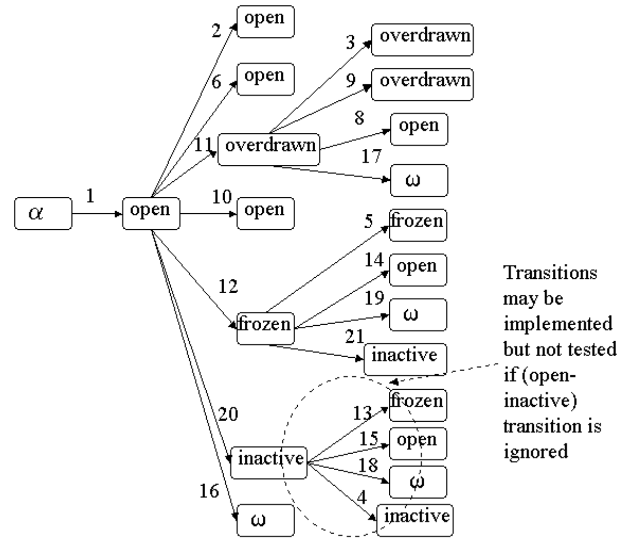

Fig. 3 Round-trip path tree of the STD example shown in Fig. 2

This introduces the need for a test case generation technique that considers all sequences of transitions which can reach each state defined in the specification. A sequence of transitions forms a path to a state. To solve this problem, we introduce a new coverage technique that ensures the coverage of all simple paths to each state in the state-transition model. The technique is called *all paths-state*.

III. All Paths-State Coverage Criterion

In the all paths-state technique, we construct a set of test cases T from a specification graph SG (e.g., UML statechart or STD of the FIC under test). T covers all simple paths to each state in the SG. A simple path includes only one iteration of a loop, if a loop exists in some sequence. Fig. 4 provides a simple visualization of the idea. The coverage criterion of the technique can be written precisely as follows:

For each state in the SG, T contains tests that traverse all simple transition sequences to the state.

The set of paths that satisfy the criterion can be shown in a tree. The procedure shown in Fig. 5 describes how to construct the tree. The procedure starts from the alpha state of the SG. In the process, whenever a state is reached the procedure traverses all the outgoing transitions from the state. The process terminates when each root-leaf tree path terminates at the omega state or a state already encountered on the path.

International Science Index, Computer and Systems Engineering Vol:2, No:4, 2008 waset.org/Publication/12038

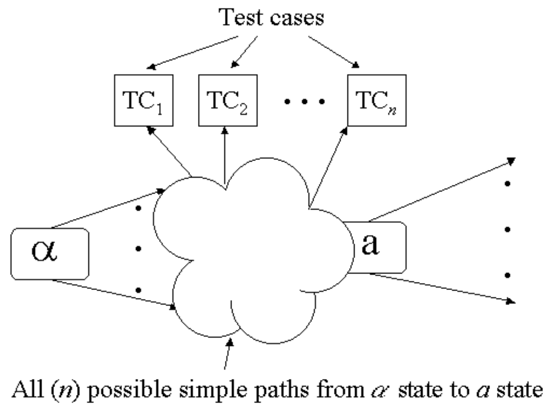

Fig. 4 All paths-state technique idea

The set of paths that satisfy the criterion can be shown in a tree. The procedure shown in Fig. 5 describes how to construct the tree. The procedure starts from the alpha state of the SG. In the process, whenever a state is reached the procedure traverses all the outgoing transitions from the state. The process terminates when each root-leaf tree path terminates at the omega state or a state already encountered on the path.

Input: A class state-based testing model
Output: The all paths-state tree of the class model.
Procedure:

1. Draw the root node of the tree to represent the alpha state.
2. Examine the state that corresponds to each *non-terminal* leaf node in the tree and each outgoing transition from the state. At least one new edge will be drawn for each transition. Each new edge and node represents an event and resultant state reached by an outgoing transition.
   a. If the transition is unguarded, the transition guard is a simple predicate, or the transition guard is a complex predicate composed of only AND operators draw one new edge.
   b. If the transition guard is a complex predicate using one or more OR operators, draw a new edge for each truth value combination that is sufficient to make the guard TRUE.
3. For each edge and node drawn in Step 2:
   a. Record the corresponding transition event, guard, and action on the new edge.
   b. If the state that the new node represents has already been encountered on the tree path that contains the new node or is the omega state, mark this node as a terminal – no more transitions are drawn from this node.

Fig. 5 Produce an all paths-state tree from a state model

Fig. 6 shows the all paths-state tree of the STD of Fig 2. In the STD, if any transition is deleted, reachable states from the deleted transition can still be reached by some other paths of the tree. For example, if all paths-state technique is used to build the test cases and the application developer chooses not to implement the transition originated from the *Open* state and ended at the *Inactive* state, the test cases that include the transition are considered broken, and therefore, they cannot be used as-is. This results in breaking the test cases built from the paths that include the transition sequences labeled as (1,20,13,21), (1,20,13,14), (1,20,13,19), (1,20,13,5), (1,20,15), (1,20,18), and (1,20,4). Note that the remaining test cases still cover all outgoing transitions from the Inactive state, and therefore, can be deployed.

Test cases are generated by traversing each path in the tree from the tree root to a leaf node. The number of generated test cases is equal to the number of leaf nodes in the tree. The number of leaf nodes in the tree shown in Fig. 6 is 22, and therefore, the number of generated test cases is 22.

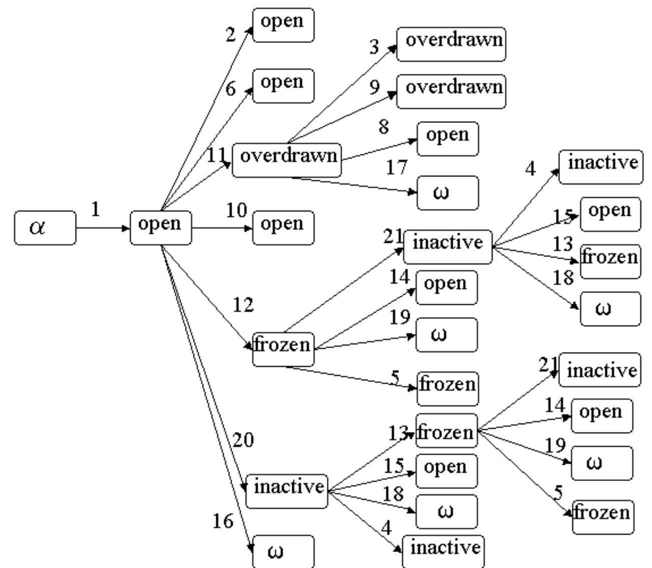

Fig. 6 All paths-state tree, constructed using the procedure shown in Fig. 5, of the STD example shown in Fig. 2

Property 1: *In terms of path coverage, the all paths-state coverage subsumes the round-trip path coverage.*
Rationale: The coverage of each of the all paths-state and round-trip path strategies is represented by a tree. The only difference between the construction procedures of the two types of trees is in the stopping criterion. In the round-trip path strategy, each path in the tree ends in either a node that represents the *omega* state in the model or a node that represents a state in the model already *represented elsewhere in the tree*. In the all-paths-state strategy, each path in the tree ends by either a node that represents the omega state in the model or a node *n* that represents a state in the model already *represented elsewhere in the path that contains node n*. As a result, the stopping criterion imposed by the all paths-state strategy is more constrained than the stopping criterion imposed by the round-trip path strategy. Consequently, each path in the round-trip path tree is identical to a sub-path in the all paths-state tree. Therefore, the all paths-state coverage subsumes the round-trip path coverage in terms of path coverage. Fig. 7 shows the path coverage hierarchy for three different strategies. The all paths-state coverage technique covers the same or more paths than the round-trip path coverage technique. The all paths-state coverage technique

International Science Index, Computer and Systems Engineering Vol:2, No:4, 2008 waset.org/Publication/12038

covers the same or less paths than the exhaustive all paths coverage criterion that covers all possible paths in a state machine.

IV. Empirical Evaluation

In this case study, we evaluate experimentally the all paths-state technique in comparison to the round-trip path and the all-transitions techniques. The comparison is performed in terms of the number of transitions covered in the updated state model after deleting transitions. Two WaveFront Pattern frameworks derived using CO_2P_3S parallel programming system [14] are considered in this study. The hooks of each framework identify two FICs and their specifications. One of the FICs is very simple (i.e., trivial), and therefore, it is not considered in this study. The characteristics of the STD of the other one, for each framework, are shown in the first row of Table I.

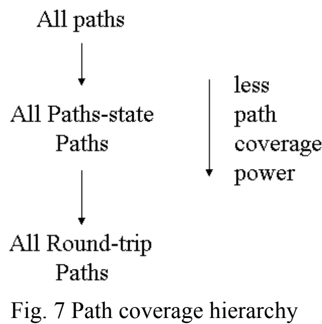

Fig. 7 Path coverage hierarchy

The case study considered three test case coverage techniques: round-trip path, all transitions, and all paths-state. The trees corresponding to the round-trip path technique and all paths-state technique are constructed from the STD of the considered FICs. For each of the two considered STDs, there are four possible different round-trip path trees that can be constructed from them using the round-trip path tree production procedure [7]. However, they all have the same number of states, nodes, and number of generated test cases and these numbers are shown in Table I. The all-transitions technique generates a test case for each outgoing transition from a state in the STD. Since some states can be reached using different paths, we have to select one path and write the corresponding test case. In our study, we followed the algorithm provided in [9] to find a path to a state. If there is more than one path to a state, the algorithm picks one of the paths. The selection of the path affects greatly the results of the case study. Therefore, in our analysis we considered each path alone, obtained the required results, and computed the average over all of the considered paths. For the transitions of the FIC in the first and second framework, 136 and 140 paths were considered, respectively. Finally, there is only one possible all paths-state tree for each STD and its characteristics are shown in the last row of Table I.

The study considered an application for each framework. The applications were not built for the purpose of the study. STDs were drawn for the implemented FICs in the applications. The names of the implemented FICs in the first and second applications are *SkylineMatrixInterface* and *MatrixBlock*, respectively. For each of the two implemented FICs, three transitions of the original STDs were removed because they are not required in modeling the specifications of the implemented FICs. This results in having 34 and 32 reused transitions in the STDs that specify the behaviors of the *SkylineMatrixInterface* and *MatrixBlock* objects, respectively. The effect of the removed transitions on the baseline test cases was analyzed for each of the three testing techniques. In the analysis, all broken baseline test cases were discarded. Finally, we counted the number of STD transitions of the *SkylineMatrixInterface* and *MatrixBlock* objects that are still covered using the non-broken test cases. Since there are four different possible round-trip path trees for each of the STDs, we have considered each round-trip path tree alone and, then, we have computed the average number of transitions covered by the non-broken test cases. We did the same analysis for the all-transitions test cases, because there are different possible paths to each state.

TABLE I
High Level Descriptions of the used Graphs

	FIC of Framework 1	FIC of Framework 2
Statechart		
No. of states	7	7
No. of transitions	37	35
Round-trip path tree		
No. of nodes	38	36
No. of edges	37	35
No. of test cases	33	33
All transitions		
No. of test cases	37	35
All paths-state tree		
No. of nodes	116	151
No. of edges	115	150
No. of test cases	95	124

Fig. 8 shows the average number of the transitions of the implemented FICs covered by the non-broken test cases when each of the three techniques is used to produce test cases for *SkylineMatrixInterface* and *MatrixBlock* classes. For example, the all paths-state coverage technique was used to generate test cases for the considered FIC of Framework 1. Table I shows that 95 test cases were required. The implementation of the FIC, i.e., *SkylineMatrixInterface* class, reused 34 out of 37 transitions introduced by the hooks. The test cases that cover the non-reused transitions (i.e., the transitions not required in modeling the specifications of the implemented FICs) were discarded because they cannot be used as-is. Fig. 8 shows that the remaining test cases were able to cover 34 reused transitions (i.e., all the reused transitions).

International Science Index, Computer and Systems Engineering Vol:2, No:4, 2008 waset.org/Publication/12038

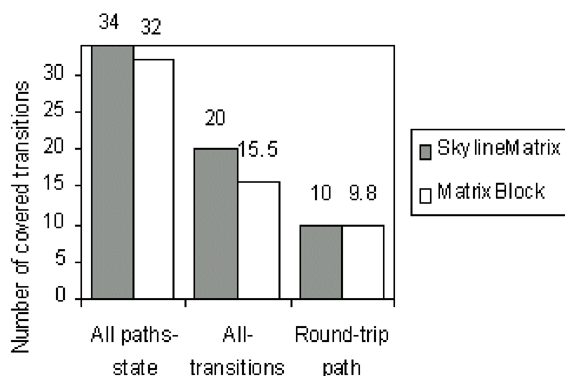

Fig. 8 Coverage comparison results

The results showed that the all paths-state technique produces test cases that are more effective than the ones produced using the all-transitions and round-trip path techniques in covering the reused transitions in the specification models of the implemented FICs. All paths-state coverage subsumes round-trip path coverage, which means that the all paths-state coverage has at least the same fault coverage effectiveness as the round-trip path one. In [16], round-trip path strategy was shown to be reasonably effective in terms of fault coverage.

V. Conclusions and Future Work

This paper helps to address the overall goal of providing a framework with effective reusable test cases. The application developer can not only use the framework design and code to build the application, but can also use the provided test cases to test part of the new application code. Part of the application code is implemented by following the hook descriptions. Hook descriptions define how to construct the FICs and introduce also the specifications of the FICs. As a result, the framework developer can produce specification-based test cases for the FICs that the application developer can use to test the implementation of the FICs.

The problem with this approach is that the application developer may implement part of the specification and decide that the rest of the specification is not required to be implemented and used in the application. This may affect the baseline test cases generated from the full specification provided through the hook descriptions. This paper addresses this problem by introducing a specification coverage criterion that produces test cases for FICs that are sufficient to cover all implemented transitions in the specification models of the FICs under test. The introduced coverage criterion is called all paths-state and it covers all paths to each state in the specification model. The criterion is experimentally evaluated. The empirical evaluation shows that all paths-state technique produces test cases with high coverage degree for the specifications of the implemented FICs comparing to test cases generated using round-trip path and all-transition techniques. A tool called FIST2 is developed to support the introduced technique. The tool reads a STD described in a tabular form, applies the all path-state coverage technique, and generated Java coded test cases.

Using all paths-state coverage technique may result in covering some transitions more than once in different paths. This might result in building redundant test cases. A redundant test case is a test case for transitions covered by one or more other test cases and no more. Redundant test cases may still exist after discarding broken test cases. Thus, it is required to investigate whether the effort in discovering the redundant test cases is less than the effort of actually applying the redundant test cases. If yes, it is required to define formally the nature of redundant test cases and develop an efficient way to detect and remove them. The test case generation technique introduced in this paper is limited to classes that have sequential behaviors. Further research is required to show how to generate test cases for classes that have concurrent behaviors.

References

[1] K. Beck and R, Johnson. Patterns generated architectures, Proc. of ECOOP 94, 1994, 139-149.
[2] G. Froehlich, H.J. Hoover, L. Liu, and P.G. Sorenson. Using Object-Oriented Frameworks, CRC Handbook of Object Technology, CRC Press, 1998, 26-1 - 26-22.
[3] G. Froehlich, H.J. Hoover, L. Liu, and P.G. Sorenson. Hooking into Object-Oriented Application Frameworks, Proc. 19th Int'l Conf. on Software Engineering, Boston, May 1997, 491-501.
[4] G. Froehlich. Hooks: an aid to the reuse of object-oriented frameworks, Ph.D. Thesis, University of Alberta, Department of Computing Science, 2002.
[5] R. Johnson and B. Foote. Designing reusable classes, Journal of Object-Oriented Programming, Vol. 2(1), 1988, pp.22-35.
[6] J. Al Dallal and P. Sorenson, Generating State-Based Testing Models for Object-Oriented Framework Interface Classes, submitted for publication in Transactions on Engineering, Computing and Technology, 2006.
[7] R. Binder. Testing object-oriented systems, Addison Wesley, 1999.
[8] T. Chow, Testing software design modeled by finite state machines, IEEE Transactions on Software Engineering, EE-4(3), 1978, 178-187.
[9] J. Offut and A. Abdurazik, Generating tests from UML specifications, Second International Conference on the Unified Modeling Language (UML99), Fort Collins, CO, October 1999, 416-429.
[10] K. Bogdanov and M. Holcombe, Statechart testing method for aircraft control systems, Software Testing, Verification and Reliability, 11(1), 2001, 39-54.
[11] A. Abdurazik, P. Ammann, W. Ding, and J. Offutt, Evaluation of three specification-based testing criteria, Sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '00), Tokyo, Japan, September 2000, 179-187.
[12] G. Antoniol, L. Briand, M. Penta, and Y. Labiche, A case Study Using the Round-Trip Strategy for State-based Class Testing, Carlton University TR SCE-01-08, revised Jan. 2002.
[13] Y. Kim, H. Hong, D. Bae, and S. Cha, Test cases generation from UML state diagrams, IEE Proc.-Software, 146(4), 1999, 187-192.
[14] S. McDonald, J. Schaeffer, and D. Szafron. Pattern-based object-oriented parallel programming. Proceedings of the First International Scientific Computing in Object-Oriented Parallel Environments Conference (ISCOPE'97), 1343 of Lecture Notes in Computer Science, 1997, 167-274.

International Science Index, Computer and Systems Engineering Vol:2, No:4, 2008 waset.org/Publication/12038