International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:5, 2008

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

Jehad Al Dallal, and Paul Sorenson

***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 Framework Interface Classes (FICs) and their possible specifications, which helps in building reusable test cases for the implementations of these classes. In applications developed using gray-box frameworks, FICs inherit framework classes or use them without inheritance. In this paper, a test-case generation technique is extended to build test cases for FICs built for gray-box frameworks. A tool is developed to automate the introduced technique.

***Keywords*—**Class testing, object-oriented framework, reusable test case.

I. Introduction

SOFTWARE testing is an important and critical verification activity considered to be a time-consuming and labor-intensive task. It aims at finding software errors in order to increase the level of confidence in development software. Central to the testing activities is the design of a test suite. The basic element of a test suite is a test case that describes the input test data, the test pre-conditions, and the expected output. A test driver is a software implementation of a test case.

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]. Frameworks are classified according to their customization method into two categories [3]: white box and black box. In white box frameworks, 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.

To build an application using a framework, application developers create two types of classes: (1) classes that use the framework classes with or without inheritance, 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 classes created by application developers. Instances of FICs are called framework interface objects. Fig. 1 shows the relationships among framework classes, hooks, and FICs. FICs use the framework classes in two ways: either by subclassing them or by using them without inheritance. Hooks define how to use the framework; 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 [4].

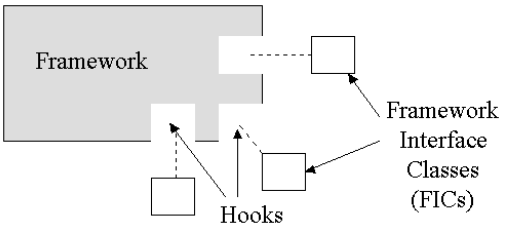

Fig. 1 Framework interface classes

Application developers can use all FICs or some of them according to their 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 a 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: α and ω, to represent the states of the object before being constructed and after being destructed, respectively. Moreover, the STD contains the Open, Overdrawn, Inactive,

Manuscript received May 12, 2007. 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 (email: Sorenson@cs.ualberta.ca)

and Frozen states to model the states of the object. A banking system developer can choose to implement the specification shown in Fig. 2 as defined. Moreover, the developer can choose to add, for example, transitions between Overdrawn and Frozen states to match the requirements of a banking system. Finally, the application developer can choose to ignore, for example, a transition originating from the Open state and ending at the Inactive state. This implies that an account can never go directly from the Open state to the Inactive state without going through the Frozen state first.

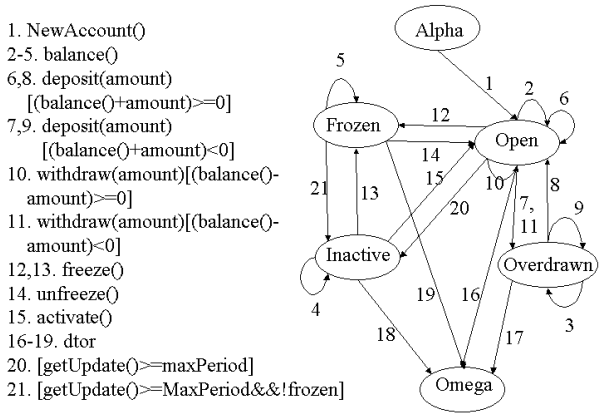

Fig. 2 The STD of the *NewAccount* object defined in the banking framework hooks.

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 everytime 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. In [5], a technique called all paths-state is introduced to build test cases for FICs. In this paper, first, it is shown that this technique is not effective in building test cases for FICs that extend framework classes. In this case, unnecessary test cases are built. Second, an effective technique is introduced to build test cases for FICs that inherit the framework classes or use them without inheritance. Building test cases and coding them is a labor-intensive work. Therefore, a tool that automates the introduced technique for Java frameworks is developed.

The paper is organized as follows. Sections II and III discuss the related work and background research, respectively. In Section IV, the extended version of the all paths–state technique is described. Section V illustrates how to generate the reusable test cases for the FICs using the extended technique. A supporting tool is introduced in Section VI. Finally, Section VII provides conclusions and a 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 [6]. 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 [6]. 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 can 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-transition 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 criterion does not put any constraints on how to reach the accepting state. Chow [7] introduced all transition coverage criterion for finite state machines and Offut et al. [8] adapted the criterion for UML statecharts and compared it experimentally with other specification coverage criteria. Bogdanov et al. [9] used all transitions coverage criterion 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 [7, 8, 10].
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 [8, 10].
4. Round-trip path coverage. In round-trip path coverage, transition sequences that start and end with the same state and simple paths from α to ω state are covered. A simple path includes only an iteration of a loop, if a loop exists in some sequence. Round-trip path strategy was proposed originally by Chow [7] and was denoted as W-method. Binder [6] adapted

the strategy to UML statecharts and called it round-trip path testing. Antoniol et al. [11] showed experimentally that the round-trip path testing strategy is reasonably effective at detecting faults. Kim et al. [12] used a criterion similar to the round-trip path strategy to derive testing trees for testing control and data flow through states.

III. Background

At the class testing level, a testing model has to be constructed and used to generate the test cases. In [4], a novel technique to synthesize a class state-based testing model from the specifications (i.e., pre-conditions and post-conditions) of class methods is introduced. Al Dallal et al. [5] introduced a test-case generation criterion called all paths-state that uses the synthesized model to generate test cases that are effective at the application testing stage in covering the reused FIC specifications.

A. Framework Hooks

In [2], the issue of documenting the purpose of a framework and how it is intended to be used using the hooks is described and formalized. Hooks describe how to extend or customize parts of the framework to build an application.

Froehlich [2] provided a special-purpose language and grammar in which the hook description can be written. Each hook description consists of the following parts. (1) a unique name, (2) the requirement (i.e., the problem the hook is intended to help solve), (3) the hook type. (4) the other hooks required to use this hook, (5) the components that participate in this hook, (6) the pre-conditions (i.e., the constraints on the parameters [or the context] that must be true before the hook can be used), (7) the changes that can be made to develop the application, (8) the post-conditions (i.e., constraints on the parameters that must be true after the hook has been used), (9) a general comment section. It is not necessary to have all the above parts for each hook.

Fig. 3 shows a hook description example for the creation of an account in a banking framework. The *Initialize Account* hook creates a constructor method for the *NewAccount* class (i.e., an FIC defined in the framework hooks). In the constructor method, the account money currency is selected. There are three pre-built classes in the framework for money: *USMoney*, *EURMoney*, and *Money*. Moreover, the user must specify the bank branches in the system. Finally, the user must specify the *maxPeriod* variable value.

The introduced hook description supports the framework application test design. The hook description identifies the FICs and their methods. In addition, it identifies the pre-conditions and post-conditions of the FIC methods. These pre-conditions and post-conditions are essential to determine the FIC behaviors and sequential constraints. Moreover, post-conditions hold the expected outputs. The pre-conditions and post-conditions of a method are called method specifications. When an FIC extends a framework class (i.e., in case of a white-box framework), the inherited methods are either used in the context of the FIC without modifications or extended. For both cases, the hook descriptions show how to use the inherited methods of the framework classes and identify their pre- and post-conditions in the context of the FICs. When an FIC uses a framework class (i.e., in case of a black-box framework), there are no methods inherited from the framework classes. In this case, the hook descriptions introduce methods for the FICs and show how to use the introduced methods. The technique proposed in [4] can be used to synthesize the class testing models for the FICs from the method specifications provided in the hooks. In addition, the method specifications can be used to evaluate the results of the test cases as proposed in [13].

Name: Initialize Account
Requirement: Initialize an account (i.e., set the currency and bank branches).
Type: Template
Uses: None
Participants: Account(framework), NewAccount(app), Amoney(app);
Pre-conditions: amount>=0;
Changes:
NewAccount.NewAccount(int amount) extends Account.Account(int amount);
Choose AM from (Money, USMoney, EURMoney);
Create Object Amoney as AM() in MyAccount. NewAccount(int);
Create Object branches as Branches() in NewAccount.NewAccount(int);
Repeat as necessary {
Acquire BranchName: string
NewAccount.NewAccount(int) -> branch.addBranch(BranchName);
}
Acquire maxPeriod : integer domains:0-999999;
NewAccount.NewAccount(int) -> NewAccount.setMaxPeriod(maxPeriod);
Post-conditions:
Operation NewAccount. NewAccount (int);
NewAccount.balance>=0;
! NewAccount.frozen;
NewAccount.getUpdate()< NewAccount.MaxPeriod
Comments:

Fig. 3 Description of the *Initialize Account* hook of a banking framework

B. All Paths-State Test-Case Generation Technique

At the application development stage, the application developer can implement part of the specification introduced by the framework hooks for FICs and decide that the rest of the specification is not required to be implemented and used in the application. This can affect the baseline test cases generated from the full specification provided through the hook descriptions. Therefore, the unaffected test cases can be insufficient to cover all implemented transitions in the specification model of the FIC under test. This problem exists when applying any of the state-based specification coverage criteria presented in Section II. In [5], the problem is solved by introducing a specification coverage criterion that produces test cases sufficient to cover all reused transitions in the

modified specification models of the implemented FICs under test. The introduced coverage criterion is called all paths-state and it is used to construct a set of test cases T from a specification graph SG (e.g., UML statechart or finite state machine of the FIC under test). T covers all simple paths to each state in the SG. A simple path includes only an iteration of a loop, if a loop exists in some sequence.

The set of paths that satisfy the criterion can be shown in a tree. The construction process of the tree starts from the α state of the SG. In the process, whenever a state is reached all outgoing transitions from the state are traversed. The process terminates when each root-leaf tree path terminates at the final (i.e., ω) state or a state already encountered on the path.

Fig. 4 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 originating from the Open state and ending at the Inactive state, the test cases that include the transition are considered broken; 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. In [5], it is proved that all paths-state coverage subsumes the round-trip path coverage, and therefore, it has at least the same error detection power.

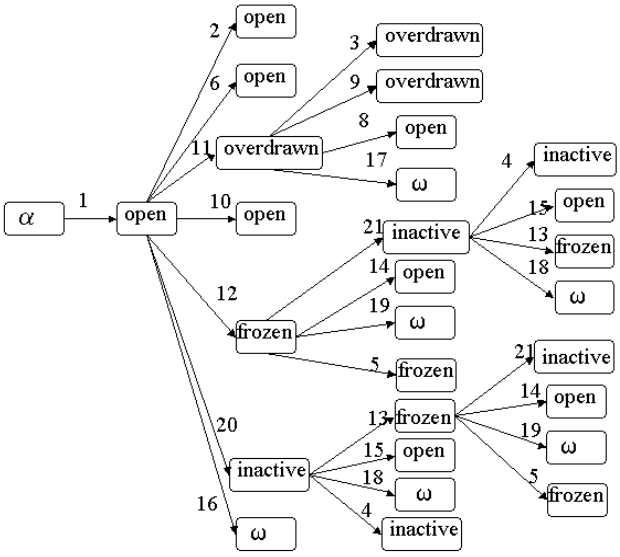

Fig. 4 All paths-state tree of the STD example shown in Fig. 2

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. 4 is 22; therefore, the number of generated test cases is 22.

IV. Extended all Paths-State Technique

Hooks can introduce FICs that subclass framework classes. This way of customizing the framework is called white-box customization. In this case, even if the application developer does not implement the FIC methods that override the inherited framework class methods, the inherited methods are accessible when the FIC is instantiated. Therefore, the specifications of the inherited methods defined in the hooks cannot be ignored. In terms of states and transitions, this results in having transitions that cannot be broken (i.e., must be implemented) at the application development stage. We call such transitions guaranteed. In Section III.B, the analysis for the NewAccount class neglects the fact that some transitions that model the class specification are guaranteed because the NewAccount class extends the Account framework class, and therefore, the application developer cannot ignore the specifications of the Account class. This produces unnecessary nodes and transitions in the all paths-state tree, as will be shown later in this section.

When the all paths-state coverage technique is applied to build test cases for the FICs that extend framework classes, some transitions can be covered using several paths. Some of the paths from the α state to the source state of the transition are composed of guaranteed transitions only. In this case, a path that contains only guaranteed transitions cannot be broken at the application development stage, and therefore, it is ineffective to cover the rest of the paths (i.e., paths that have not-guaranteed transitions) from α state to the source state of the considered transition. For example, for the STD shown in Fig. 2, if the transitions from the α state to the open state and from the open state to the inactive state are guaranteed, the outgoing transitions from the inactive state are guaranteed to be reached by following the path of the guaranteed transitions. Since this path cannot be broken at the application development stage, there is no need to cover the other paths from the α state to the inactive state in the all paths-state tree to ensure the coverage of the outgoing transitions from the inactive state.

In the state-transition diagram, if a path to a state has all transitions marked guaranteed, we say that the state has a guaranteed path. In the all path-state tree, a state can be represented by more than one node because we have to cover all simple paths to it such that if one path to a state has a transition not used by the application developer, the state remains reachable in the tree using other paths. Outgoing transitions from a state that has a guaranteed path do not have to be covered in the tree using other paths, which reduces the tree complexity. The procedure given in Fig. 5 shows how to construct the all paths-state tree from a state-transition model that has guaranteed transitions.

The procedure starts from the root state of the state-transition model. In the process, whenever a state is reached the procedure traverses all outgoing transitions from the state.

The procedure marks the tree nodes that have guaranteed paths as guaranteed nodes. The procedure traverses the outgoing transitions from the states represented by guaranteed nodes before the outgoing transitions from the other states. The process terminates when each root-leaf tree path ends at the ω state, a state represented previously in the path, or a state represented previously in the tree by a guaranteed node.

Input: A class state-based testing model that has guaranteed transitions.
Output: The all paths-state tree of the class model.
Procedure:
1. Draw the root node of the tree to represent the *α* state. Mark the node as *non-terminal* and *guaranteed*.
2. Search for a state that corresponds to a *non-terminal guaranteed* leaf node in the tree. If none is found, search for a state that corresponds to a *non-terminal not-guaranteed* leaf node in the tree.
3. Examine each outgoing transition from the state. At least one new edge will be drawn for each outgoing transition from the state. 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 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 branch for each truth value combination that is sufficient to make the guard TRUE.
4. For each edge and node drawn in step 3:
   a. Note the corresponding transition event, guard, action, and guarantee information on the new edge.
   b. If the edge and its source node in the tree are marked *guaranteed*, mark the destination node of the edge as *guaranteed*. Otherwise, mark it as *not-guaranteed*.
   c. If the state that the new node represents is the *ω* state, the state is already represented by another node (in the path containing the new node), or the state is represented somewhere else in the tree by a *guaranteed* node, mark this node as a terminal – no more transitions are drawn from this node. Otherwise, mark it as non-terminal.
5. Repeat steps 2, 3, and 4 until all leaf nodes are marked terminal.

Fig. 5 Produce an all paths-state tree from a state model that includes guaranteed transitions.

For example, suppose that the transitions that are necessary to implement the open and inactive states (i.e., transitions labeled by 1, 2, 4, 6, 10, 15, 16, 18, and 20) shown in Fig. 2 are introduced by the Account class, which is a framework class. The rest of the transitions are not defined in the Account class, but they are defined in the hooks. In this case, the transitions labeled by 1, 2, 4, 6, 10, 15, 16, 18, and 20 are guaranteed transitions. When the procedure shown in Fig. 5 is applied, the tree shown in Fig. 6 is constructed.

In the construction process, first, the root node represents the *α* state of the STD. In the first iteration of the repeat loop of the procedure, an edge and a node are drawn to represent the outgoing transition from the *α* state and the transition destination state (i.e., *Open* state). The two nodes drawn in the first iteration are bolded. A bolded node and a bolded edge represent a guaranteed node and a guaranteed edge, respectively. The *α* node is always marked *guaranteed*. The outgoing edge from the *α* node is bolded because it represents a guaranteed transition. Finally, the *open* node is marked *guaranteed* because it is a destination node of a guaranteed edge initiated from a guaranteed node.

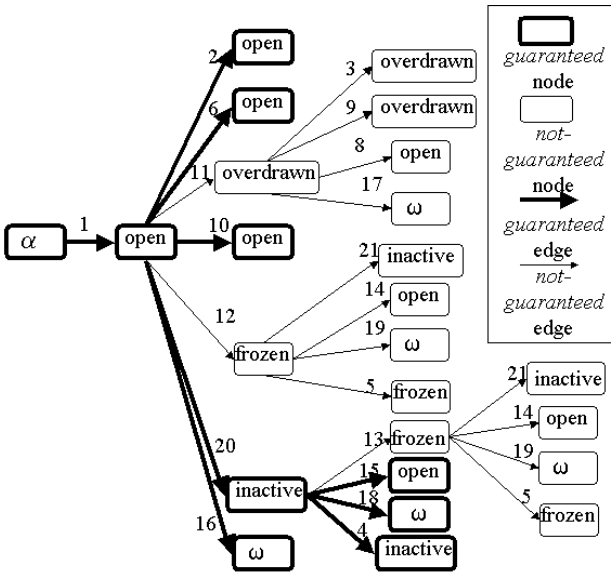

Fig. 6 All paths-state tree of the STD example shown in Fig. 2, constructed using the procedure shown in Fig. 5

In the second iteration of the repeat loop, all the outgoing transitions from the Open state are added to the tree. This includes adding the edges labeled as 2, 6, 11, 10, 12, 20, and 16, and adding all the nodes reached by these edges. In the tree drawn so far, nodes reached by the edges labeled by 2, 6, 10, and 16 are marked terminal (i.e., no more edges are drawn from them) because they are either previously encountered on the tree paths that contain them or represent the ω state. In addition, nodes reached by the edges labeled by 2, 6, 10, 20 and 16 are marked guaranteed because they are destination nodes of guaranteed edges initiated from a guaranteed node.

The tree drawn so far contains one node marked as non-terminal and guaranteed, which is the node that represents the Inactive state. Therefore, in the third iteration of the repeat loop, edges that represent the outgoing transitions from Inactive state and the nodes that represent the states reached from the Inactive state are drawn. This includes adding the edges labeled as 13, 15, 18, and 4, and adding the nodes reached by these edges. Three of the drawn nodes in the third iteration (i.e., open, ω, and inactive) are marked terminal because they are either previously encountered on the tree paths that contain them or represent the ω state. In addition, these nodes are marked guaranteed because they are destination nodes of guaranteed edges initiated from a guaranteed node. The fourth node (i.e., frozen) is marked non-

terminal and not-guaranteed.

All the non-terminal leaf nodes drawn so far are marked not-guaranteed. Therefore, in the fourth iteration we can pick any of the states that represent each of them. In this example, the node that represents the overdrawn state is picked. In the fourth iteration, edges that represent the outgoing transitions from the overdrawn state and the nodes that represent the states reached from the overdrawn state are added to the tree. This includes adding the edges labeled 3, 9, 8, and 17, and the nodes reached by these edges. All the drawn nodes are marked terminal because they are either previously encountered on the tree paths that contain them or represent the ω state. In addition, these nodes are marked not-guaranteed because they are reached by not-guaranteed edges.

All the non-terminal leaf nodes drawn so far are marked not-guaranteed. Therefore, in the fifth iteration, any of the states that represent each of them can be picked. In this example, the node that represents the frozen state reached by the edge labeled as 12 is picked. Edges that represent the outgoing transitions from the frozen state and the nodes that represent the states reached from the frozen state are added to the tree. This includes adding the edges labeled as 21, 14, 19, and 5, and the nodes reached by these edges. Three of the drawn nodes (i.e., open, ω, and frozen) are marked terminal because they are either previously encountered on the tree paths that contain them or represent the ω state. The fourth drawn node (i.e., inactive) is marked terminal because it represents a state represented in the tree by a guaranteed node (i.e., the state represented by the node at the end of path (1->20)). The four nodes are marked not-guaranteed because they are reached by not-guaranteed edges.

So far, only one leaf node in the tree is marked non-terminal. Edges that represent the outgoing transitions from the frozen state and the nodes that represent the states reached from the frozen state are added to the tree in the sixth iteration. This includes adding the edges labeled 21, 14, 19, and 5, and the nodes reached by these edges. All the drawn nodes are marked terminal because they are either previously encountered on the tree paths that contain them or represent the ω state. In addition, these nodes are marked not-guaranteed because they are reached by not-guaranteed edges. After the sixth iteration, all the leaf nodes are marked terminal, which indicates that the construction process of the tree is completed.

V. Generating Reusable Test Cases

The constructed all path-state tree can be used to build reusable test cases for the FICs. This section discusses how to build the reusable test cases and how to implement them.

A. Generating the Test Cases

The procedure given in Fig. 7 shows how to generate the test cases from the all paths-state tree that has guaranteed nodes. The test cases are generated in two rounds. In the first round, each path from the root node to a leaf node is used to build a test case. The number of test cases built in this round is equal to the number of leaf nodes. In the second round, we search for all non-terminal nodes marked as guaranteed that have all outgoing edges marked as not-guaranteed. For each of these nodes, we build a test case that traverses the path from the root node to the node marked guaranteed. This round is necessary because the application developer can decide not to use any of the methods associated with the outgoing edges from the state. In this case, all the test cases built from the paths that include the unused edges are considered broken. This results in having no test cases to test the transitions that have their destination states represented by guaranteed nodes in the tree. Fig. 8 depicts the problem. The node labeled by C is a non-terminal guaranteed node and all its outgoing edges are marked as not-guaranteed. In the first round of generating test cases, three test cases are generated to cover the paths (A->B), (A->C->D), and (A->C->E). If the application developer decides not to use the methods associated with the edges (C->D) and (C->E), the test cases generated from the paths (A->C->D), and (A->C->E) will be broken. Therefore, the edge (A->C) is not going to be covered by the remaining test case. To overcome this problem, we have introduced the second round. In the second round, the path (A->C) is used to build an additional test case.

Input: All paths-state tree that has guaranteed nodes and edges.
Output: The test cases generated from the all paths-state tree.
Procedure:

1. *for* each path from the root node to a leaf node in the all paths-state tree *do*
   Build a test case that traverses the path.
2. Search for all non-terminal nodes in the tree marked *guaranteed* and that have all their outgoing edges marked as *not-guaranteed*.
3. *for* each node *n* found in Step 2 *do*
   Build a test case that traverses the path from the root node to node *n*.

Fig. 7 Generate test cases from the all paths-state tree that has guaranteed nodes

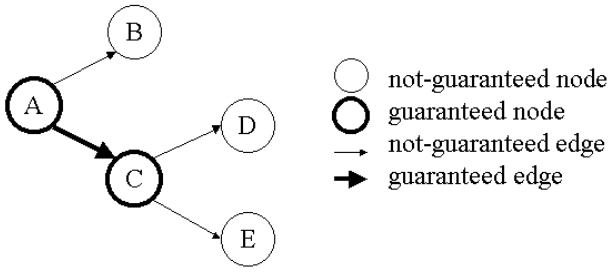

Fig. 8 Non-terminal guaranteed node special case

Finally, it is important to note that the introduced technique can be used for both the FICs that inherit the framework classes and the FICs that use the framework classes without inheritance. For the latter case, all the transitions in the STD are declared not-guaranteed. The resulting all paths-state tree, in this case, is going to be similar to the one built using the original all paths-state technique. As a result, the extended

technique can be used to generate test cases for gray-box framework interface classes.

B. Generating Test Drivers

In the previous section, it is shown how to select the sequences of message executions (i.e., sequences of transitions that form the all paths-state paths) to be tested. Each sequence of message executions forms a test case. To automate the testing process, it is required to generate test drivers (i.e., implementations of the test cases). To execute any message associated with a transition, it is required to set test values for the parameters of the message and to execute the code required to satisfy the predicates of the transition. The general problem of the automatic generation of the test values for the parameters of the message and the automatic generation of the code required to satisfy the predicates of the transition is considered future work. In this work, the required code for the test values and predicates is associated manually with the transitions of the testing model.

After executing a transition, it is required to check whether the actions associated with the transition are performed correctly and whether the transition leads to the expected resulting state. Once the testing model is synthesized using the technique introduced in [4], the transition actions and the state-invariants are associated with the transitions and states of the model, respectively. When the test drivers are developed, the checking of the code for the actions and state-invariants are instrumented into the test drivers and executed at run time to evaluate the test cases.

There are two ways to implement the test cases: either to implement all of them in one class or to implement each test case in a separate class. At the application development stage, the test drivers that can be reused to test an implemented FIC are determined. These test drivers are a subset of the test drivers provided with the framework to test the FIC. If all test drivers are included in one class, the non-applicable test drivers provided with the framework would be included in the class of the test drivers and not used, which is an ineffective solution. Therefore, it is better to implement each test driver in a separate class. This allows the application developer to maintain only the test drivers that implement the applicable test cases instead of maintaining all the test drivers provided with the framework.

The procedure given in Fig. 9 shows how to construct the test drivers for selected paths of a testing model. The procedure implements a class for each test case. Each class includes a constructor method. When the constructor method is invoked at test time, the actual testing is performed. In the constructor method, the code for executing the sequence of message executions is listed. For each message associated with a transition, the code sets up the parameter values and includes the statements required to satisfy the transition predicates. The setting-up code is followed by the message invocation statement and checking statements for the resulting actions and the state-invariants of the resulting state.

In this paper, the examples used are coded in Java language; however, the introduced techniques are applicable for frameworks and applications written in any other object-oriented language. Fig. 10 shows two Java test driver examples generated from the tree shown in Fig. 6. The two test cases are generated by traversing the paths that include the transition sequences labeled as (1->2) and (1->12->14), respectively. The checking statements for the actions and the state-invariants are written as Javadoc comments using the Design-by-Contract (DbC) language [14]. These Javadoc comments are translated at compilation time into Java code using a tool called Jcontract [15]. At run time, the Jcontract tool checks the Java statements translated from the DbC statements and reports any violations. Appendix B shows all the test drivers for the NewAccount FIC generated using the supporting tool introduced in Section VI.

Inputs: Paths in the FIC state-based model required to implement the test cases.
Outputs: FIC test drivers.
Procedure:
for each path required to implement a test case *do*
  Create a new file
  Create a class for the test driver in the file.
  Create a constructor method in the class.
  s is the first state in the path.
  Repeat
    transition *t* is the outgoing transition from state *s* in the path
    if the method invoked by the transition *t* has parameters *then* add the code require to set the test values of the parameters to the code of the constructor method.
    if the transition *t* has predicates *then* add the code required to satisfy the predicates to the code of the constructor method.
    if the transition *t* is the first transition in the path *then* insert a creation statement in the constructor method for the instance of the FIC for which the reusable test drivers are constructed.
    else insert a method call statement in the constructor method for the event associated with the transition.
    if the transition *t* has actions, insert statement(s) in the constructor method to check whether the actions associated with the transition are performed.
    Insert statement(s) in the constructor method to check whether the invariants of the reached state by the transition *t* are satisfied.
    s is the destination state of the transition *t*.
  until state *s* is the last state in the path.

Fig. 9 Construction procedure of the test drivers

VI. Automation

Framework Interface State Transition Tester ($FIST_2$) is a tool that supports the generation of the reusable test drivers for Java framework FICs at the framework development stage. It also deploys, executes, and evaluates the test drivers at the application development stage. In this paper, only the role of the tool in generating the reusable test drivers is presented.

Test Case # 1 (covers transition sequence 1->2)

```
public class TEST1_NewAccount{
    public TEST1_NewAccount(){
    /* testing the transition labeled as "1" */
    /* code for setting the parameter value  */
        float amount=1;
         /* invoking the message associated with
         the transition */
        NewAccount o = new NewAccount(amount);
       /* DbC checking statement for the
       invariants of the resulting state: Open*/
        /** @assert((o.balance()>=0) && ((o.
       getCurrentDate()-o.getLastActivityDate())<
       o.getMaxPeriod()) && !(o.isFrozen())) */

    /* testing the transition labeled as "2" */
    /* invoking the message associated with the
       transition */
        o.balance();
       /* DbC checking statement for the
       invariants of the resulting state: Open */
        /** @assert((o.balance()>=0) && ((o.
       getCurrentDate()-o.getLastActivityDate())<
       o.getMaxPeriod()) && !(o.isFrozen())) */
    }
}
```

Test Case # 9 (covers transition sequence 1->12->14)

```
public class TEST9_NewAccount{
    public TEST9_NewAccount(){
    /* testing the transition labeled as "1" */
    /* code for setting the parameter value  */
        float amount=1;
         /* invoking the message associated with
         the transition */
        NewAccount o = new NewAccount(amount);
         /* DbC checking statement for the
         invariants of the resulting state:Open*/
        /** @assert((o.balance()>=0) && ((o.
       getCurrentDate()-o.getLastActivityDate())<
       o.getMaxPeriod()) && !(o.isFrozen())) */

    /* testing the transition labeled as "12" */
    /* invoking the message associated with the
       transition */
        o.freeze();
       /* DbC checking statement for the
       invariants of the resulting state:Frozen*/
        /** @assert((o.balance()>=0) &&
       ((o.getCurrentDate()-o.getLastActivityDate
       ())< o.getMaxPeriod())&&(o.isFrozen())) */

    /* testing the transition labeled as "14" */
    /* invoking the message associated with the
       transition */
        o.unfreeze();
         /* DbC checking statement for the
         invariants of the resulting state:Open*/
        /** @assert((o.balance()>=0) &&
       ((o.getCurrebtDate()-o.getLastActivityDate
       ())< o.getMaxPeriod())&&!(o.isFrozen()))*/
    }
}
```

Fig. 10 Two test case examples generated from the tree shown in Fig. 6

At the framework development stage, $FIST_2$ tool supports the generation of the reusable test drivers for Java framework FICs. The tool semi-automates the construction of the state-transition tables for the FICs, checks the correctness of the tables, and generates reusable test drivers using the extended all paths-state technique.

Fig. 11 shows the high-level design of the tool when used at the framework development stage. The user (typically the framework developer in a test-case generation role) selects the framework. The framework is stored in a database that contains the framework code and the descriptions of the hooks. The tool passes the hook descriptions to the FIC state-transition table builder module. The FIC state-transition table builder module parses the pre-conditions and post-conditions of the FIC methods, analyzes them, and produces the state-transition table for the FIC. The framework developer can edit the generated table to add the code required to satisfy the predicates of the transitions and to add the non-event-driven transitions. The tool translates the tabular form of the state-transition model into text and stores the text in a file in the framework database. The user can use the Model Checker module of $FIST_2$ tool to check the existence of one entry and one exit state in the state model and that all the states are reachable from the entry state.

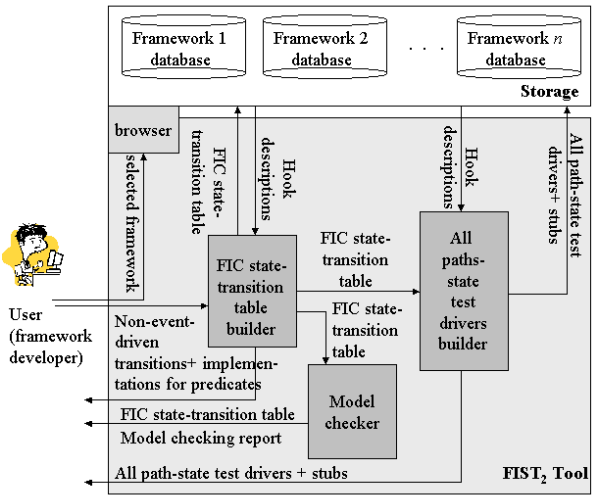

Fig. 11 The high-level design of $FIST_2$ tool (framework development stage)

The All paths-state test drivers builder component of $FIST_2$ tool uses the state-transition table to generate the all paths-state test drivers and associates the test driver identifiers with the model transitions. In addition, the tool uses the hook descriptions to determine and generate the stubs required at the application testing stage to isolate the FICs. The test drivers and stubs are stored in the framework database and provided to the user.

In the prototype version of the tool, the method specifications are not extracted from the hooks automatically because the module responsible for constructing the state-transition model from the method specifications is not yet implemented. Instead, the user has to construct the model manually from the method specifications listed in the hooks, using the algorithms provided in [4]. The user is, however, provided with a friendly GUI to input the model description in a tabular form. When entering the model description, the user specifies the guaranteed and not-guaranteed transitions. In

addition, the prototype version of the tool does not create the required stubs. However, the extended all paths-state technique is fully implemented in the prototype version of the tool.

VII. 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 can 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 can affect the baseline test cases generated from the full specification provided through the hook descriptions. All paths-state technique solves this problem. However, it is limited for FICs that do not inherit framework classes. In this paper, the technique is extended such that it can be used in generating reusable test cases for white- and gray-box frameworks, as well as the black-box ones. The extended technique is automated in a tool called $FIST_2$. The usefulness of the tool is shown by applying it to an example illustrated in this paper.

The introduced technique is limited to classes that have sequential behaviors. It can be extended to consider the classes that have concurrent behaviors. In future, we plan to extend the tool to automate the generation of the class testing models and the stubs.

References

[1] Beck and R, Johnson. *Patterns generated architectures*, Proc. of ECOOP 94, 1994, pp. 139-149.
[2] G. Froehlich. *Hooks: an aid to the reuse of object-oriented frameworks*, Ph.D. Thesis, University of Alberta, Department of Computing Science, 2002.
[3] R. Johnson and B. Foote. *Designing reusable classes*, Journal of Object-Oriented Programming, Vol. 2(1), 1988, pp. 22-35.
[4] J. Al Dallal and P. Sorenson, *Generating State-Based Testing Models for Object-Oriented Framework Interface Classes*, Transactions on Engineering, Computing and Technology, Vol. 16, 2006, pp. 96-102.
[5] J. Al Dallal and P. Sorenson, *Generating Class-Based Test Cases for Interface Classes of Object-Oriented Black Box Frameworks*, Transactions on Engineering, Computing and Technology, Vol. 16, 2006, pp. 90-95.
[6] R. Binder. *Testing object-oriented systems*, Addison Wesley, 1999.
[7] T. Chow, *Testing software design modeled by finite state machines*, IEEE Transactions on Software Engineering, EE-4(3), 1978, pp. 178-187.
[8] J. Offut and A. Abdurazik, *Generating tests from UML specifications*, Second International Conference on the Unified Modeling Language (UML99), Fort Collins, CO, October 1999, pp. 416-429.
[9] K. Bogdanov and M. Holcombe, *Statechart testing method for aircraft control systems, Software Testing, Verification and Reliability*, 11(1), 2001, pp. 39-54.
[10] 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, pp. 179-187.
[11] 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.
[12] Y. Kim, H. Hong, D. Bae, and S. Cha, *Test cases generation from UML state diagrams*, IEE Proc.-Software, 146(4), 1999, pp. 187-192.
[13] L. Briand, Y. Labiche, *A UML-based approach to system testing*. Technical Report TR SCE-01-01. Carlton University: Canada, 2002.
[14] B. Meyer, Design by contracts, IEEE Computer, 1992, Vol. 25(10), pp. 40-52.
[15] Jcontract,http://www.parasoft.com/jsp/products/home.jsp?product=Jcontract, ParaSoft Corporation, July 2006.