Testing Software Assets of Framework-Based Product Families during Application Engineering Stage

Jehad Al Dallal
Kuwait University/Department of Information Sciences, Kuwait
Email: jehad@cfw.kuniv.edu

Paul Sorenson
University of Alberta/Department of Computing Science, Edmonton, Alberta
Email: sorenson@cs.ualberta.ca

***Abstract*— An application framework provides reusable design and implementation for a family of software systems. At the application engineering stage, application developers extend framework assets to build their particular framework instantiations. Typically, framework software assets are tested before being used. However, achieving complete coverage of a system under test is impossible or at least impractical. Therefore, framework software assets can have undiscovered errors that appear in some instantiations. During the application engineering stage, it is important to identify the framework use cases that are used in the instantiation but not covered during the framework testing stage.**

In this paper, a testing model that considers retesting framework assets during the application engineering stage is proposed. In addition, a test-case-reusing technique is introduced to identify uncovered framework use cases and cover them by reusing the test cases already built during the framework domain engineering stage. Empirical studies are reported to show the adequacy of the proposed framework test-case-reusing technique in terms of reducing testing time and effort, and a supporting tool that automates the proposed test-case-reusing technique is developed and introduced.

***Index Terms*— hooks, object-oriented framework, domain engineering, application engineering, object-oriented framework instantiation testing, test case reusability**

I. Introduction

Software engineering aims at developing techniques and tools that reduce the software development time while simultaneously enhancing the product quality. Using object-oriented frameworks is an appealing way to speed up development of a software product family [1]. A software product family is a set of software products that share common features [2]. An application framework is the reusable design and implementation of a system or subsystem [3]. It contains a collection of reusable concrete and abstract classes. These classes are referred to in this paper as framework assets. The framework design provides the context in which framework assets are used. The framework itself is not complete. Object-oriented framework engineering is divided into separate domain and application engineering tasks. During domain engineering, the framework assets are produced. During application engineering, the users of the framework complete or extend the framework assets to build their particular instantiations instead of developing the applications from scratch. Design for reusability is a costly and time-consuming task. However, reusing the framework design and code reduces application development time and cost considerably. As a result, there is a high probability that the cost and time spent during the framework domain engineering stage will be recouped after producing a few framework applications.

Typically, reusing test cases instead of creating them from scratch reduces testing time and effort. To use reusable test cases for testing a specific application, a mechanism is required to identify the applicable test cases. For the applicable test cases, a technique is required to identify whether the test cases can be used as-is or must be modified or extended. If the test cases have to be modified or extended, the way to perform such modification or extension should be easy and straightforward. The identification and use of reusable test cases must be systematic and fully or at least semi-automated. The cost of reusing the test cases must be much lower than the cost of building the test cases from scratch; otherwise, application developers will prefer to build their own test cases.

Testing the framework during the domain engineering stage is essential. If the framework code contains errors, the errors will be passed on to the instantiations developed from the framework. Several techniques have been proposed to test frameworks during the framework development stage (e.g., [4], [5], and [6]). In these testing techniques, different possible framework use cases and input data are exercised. Tevanlinna et al. [1] mention that during the application engineering stage, it is hard to rely on the testing performed during the framework

© 2008 ACADEMY PUBLISHER

domain engineering stage. When the framework is instantiated, some framework input data and application-specific framework-use-cases not covered during the framework testing stage can be applied by the framework instantiation. When these input data and use cases are applied, some undetected framework errors can appear and cause the framework instantiation to function improperly. Therefore, it is important to reconsider framework testing during the application engineering stage. Solving this problem requires resolving the following two issues:

1. Identifying the framework input data and use cases that are used in the instantiation and not covered during the framework domain engineering stage.
2. Proposing an adequate test-case-reusing technique to test the uncovered framework input data and use cases. The technique reuses the testing assets (e.g., test suites and testing models) used during the framework testing stage, which reduces the testing effort.

This paper addresses the above two issues. The traditional framework testing process (e.g., [4], [5], and [6]) is performed during the domain engineering stage. In this process, demo applications and test assets are created and used to test the framework. The framework is not further tested at the application engineering stage. In this paper, the framework testing process model is extended to consider testing the application-specific framework input data and use cases that are not covered during the domain engineering stage. In the extended model, when the classes that use the framework classes are developed during the application engineering stage, the framework test input data and use cases not covered by the framework test assets are detected. The classes that use the framework classes are called Framework Interface Classes (FICs) [7]. The framework test assets are customized and reused to retest the framework and cover the nontested input data and use cases. The customized test cases are added to the framework database such that at any time the framework database contains the test assets created during the framework domain engineering task and the test assets added during the application engineering stage of the previously developed instantiations. This incremental model reduces the framework testing effort each time a new framework instantiation is developed. In addition, the extended model increases confidence in the framework as more instantiations are developed. The extended framework testing process model is shown in Fig. 1. In this figure, the testing process performed at the application engineering stage models the actual extension to previous work.

The technique introduced in this paper extends a framework testing technique called Testing Framework Through Hooks (TFTH) [5]. First, the paper explains how to identify the framework input data and use cases that are used in the instantiation and not covered during the framework testing stage. Second, the paper introduces a test-case-reusing technique that considers the reusability of the framework testing assets to build test cases that test the used portion of the framework. Experiments applying the introduced technique to two framework instantiations were conducted. The experiments show the adequacy of the proposed framework test-case-reusing technique in terms of reducing testing time and effort. As in [8], the technique introduced in this paper assumes that the framework test assets are provided with the framework. A supporting tool called JFramework Re-Tester is developed to automate the introduced testing technique.

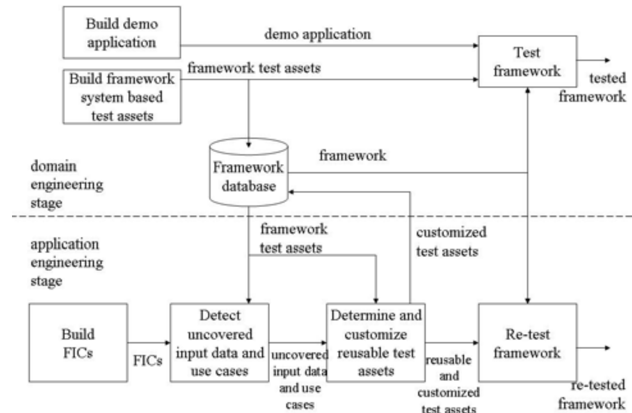

Figure 1. The extended framework testing process model

The paper is organized as follows. Section II discusses background and related research. Section III introduces the test-case-reusing technique that reuses test cases to test the framework part of the instantiation. An experiment showing the adequacy of the introduced test-case-reusing technique in terms of reducing testing time and effort is described in Section IV. Section V discusses the automation of the proposed test-case-reusing technique. Finally, Section VI provides conclusions and a discussion of future work.

II. Background and Related Research

Testing Frameworks Through Hooks (TFTH) [5] is a testing technique that tests hook-documented frameworks at system level. This section gives an overview of object-oriented framework paradigm, TFTH technique, and other related work.

A. *Object-Oriented Framework Paradigm*

An application framework provides a reusable design and implementation for a family of software systems [3]. During application engineering stage, users of the framework complete or extend the framework assets to build their particular instantiations. When the framework is used during the application engineering stage, developers build 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 FICs because they act as interfaces between the framework classes and the second type of classes created by application developers. Places at which users can add their own classes are called hook points [9, 10].

In [9] and [10], the issue of documenting the purpose of a framework and how it is intended to be used with hooks is described and formalized. The concept of hook

© 2008 ACADEMY PUBLISHER

description is introduced. The hook description explains how to extend or customize part of the framework to build an instantiation. Froehlich [10] provides a special-purpose language and grammar in which the hook description can be written. Each hook description consists of the following: (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 preconditions (i.e., 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 instantiation, (8) the postconditions (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. 2 shows a hook description example for creation of an account in a banking framework. The user of the framework creates a *NewAccount* class that subclasses the *Account* class which is included in the framework. The hook *Initialize Account* creates an *init* method that should be included in *NewAccount* class and called when an account is constructed. In *init* method, the account currency is selected. There are three prebuilt classes in the framework for money: *USMoney*, *EURMoney*, and *Money*. Moreover, the user has to specify the bank branches in the system.

Name: Initialize Account
Requirement: Initialize an account.
Type: Template
Uses: None
Participants: Account(framework), NewAccount(app), Amoney(app);
Preconditions: Subclass NewAccount of Account;
Changes: New operation NewAccount.init();
Choose AM from (Money, USMoney, EURMoney);
Create Object Amoney as AM() in MyAccount.init();
Create Object branches as Branches() in NewAccount.init();
Repeat as necessary {
Acquire BranchName: string
NewAccount.init() -> branch.addBranch(BranchName);}
Postconditions: Operation NewAccount.init();
NewAccount.branches!=Null;
Comments:

Figure 2. Description of the *Initialize Account* hook of a banking framework

B. TFTH Framework Testing Technique

TFTH is a testing technique that tests frameworks at system level. It tests that the framework use cases are implemented correctly. FICs extend or use framework classes to implement the use cases; therefore, TFTH tests frameworks through FICs, which are composed of hook methods. TFTH tests the framework use cases by enacting the hook methods through which the framework use cases are performed. Hook descriptions specify the behaviors of the FICs, and they are used to construct the FIC Hook State Transition Diagram (HSTD) automatically [7]. The HSTD models FIC behavior and consists of nodes and direct links. Each node represents a state (i.e., a set of instance-variable value combinations of the class object) and each link represents a transition caused by an event. There are two types of links: solid and dotted, which represent transitions associated with explicit and implicit events, respectively. Implicit events are calls to methods called implicitly by other methods. The transition labels have the following form:

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

Fig. 3 shows the HSTD of an *account* banking FIC. The HSTD contains two special states: alpha and omega, to represent the states of the object before being constructed and after being destructed, respectively. Moreover, the HSTD contains the *Open, Overdrawn, Inactive,* and *Frozen* states to model the rest of the states of the object. Since the event *NewAccount()* invokes the *init* event defined in the banking framework hooks, an *initializing* state is added. Moreover, a transition (represented by dotted link) associated with the implicit call of *init* is added from the *initializing* state to the *Open* state.

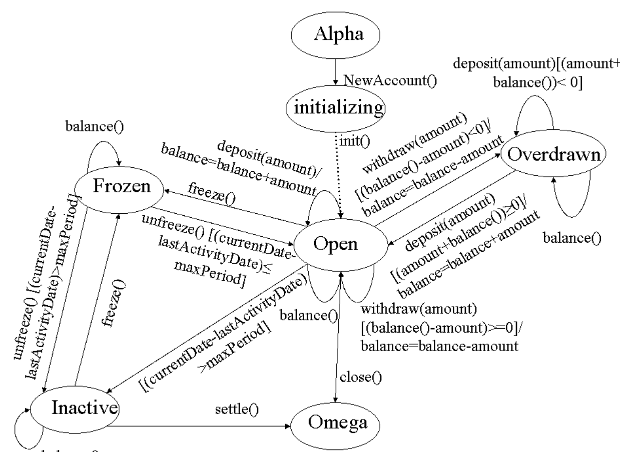

Figure 3. HSTD of the *NewAccount* object defined in the banking framework hooks

Hook descriptions define how to construct FIC methods. These methods are called *hook methods*. For example, in the *changes* section of the *Initialize Account* hook (Fig. 2), the way to construct the *init* method is described.

Each hook method (i.e., a method defined in hook descriptions) is modeled by a Construction Flow Graph (CFG), a graphical representation of the control structure of the construction sequence of the hook method contents. Fig. 4 shows the CFG of the *init()* method described in the *Initialize Account* hook (Fig. 2) of the banking framework. The hook statement '*Create Object Amoney …*' is represented by three nodes because there are three possible framework *money* classes: *Money*, *USMoney*, and *EURMoney*.

Typically, FICs consist of multiple hook methods. Each hook method introduced in the hook description can have different possible implementations. Therefore, each FIC can have multiple different possible implementations. The FIC implementations are constructed by considering the combinations of possible implementations of hook methods. A demo instantiation that can be used in the testing process consists of an

© 2008 ACADEMY PUBLISHER

implementation of one or more FICs and the framework code.

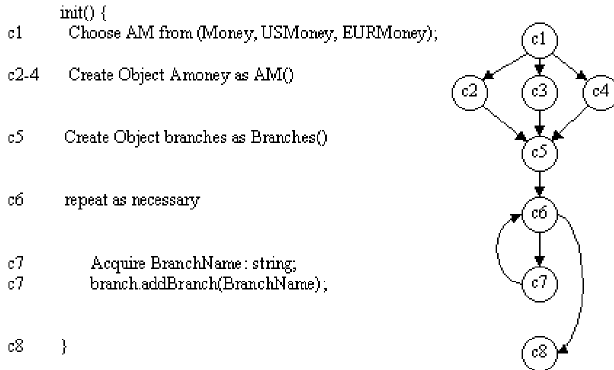

Figure 4. The CFG of the *init()* method defined in the *Initialize Account* hook

TFTH technique generates a framework test suite automatically in seven steps, as follows:

1. Determine the FICs.
2. Construct the HSTDs of the FICs.
3. Produce a round-trip path tree for each HSTD using Binder's procedure [4]. 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. Fig. 5 shows the round-trip path tree of the HSTD shown in Fig. 3.
4. Construct a CFG for each hook method.
5. Generate test data for all parameters of the hook methods.
6. Generate hook method possible implementations. Each implementation of a hook method exercises a combination of test data, generated in Step 5, in a CFG *simple complete path* or *extreme complete path*. A *complete path* is a path that starts at the graph's entry node and ends at the graph's exit node. A *simple complete path* is a complete path that includes at most an iteration of a loop, if a loop exists in some sequence. An *extreme complete path* is a complete path that includes at least (maximum number of iterations of a loop –1), if a loop exists in some sequence.

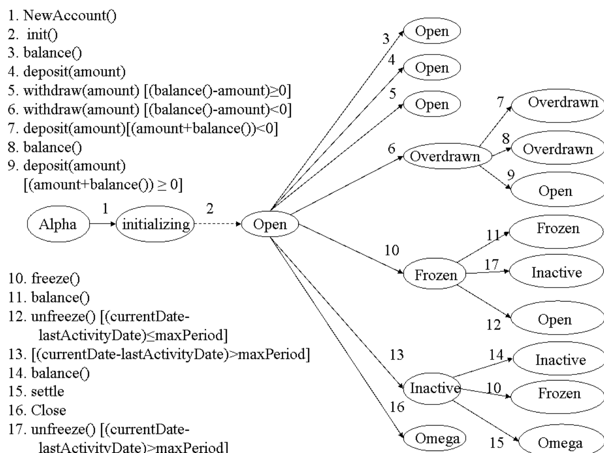

Figure 5. The round-trip path tree of the HSTD shown in Fig. 3

7. Produce framework test cases. Each test case exercises a single round-trip path and covers one possible combination of implementations of hook methods called in the round-trip path. A complete framework test suite contains test cases that cover all combinations of parameter test data in all simple and extreme complete paths of the CFGs and simple complete round-trip paths in all round-trip path trees.

C. *Other Related Work*

Several recent research studies address the problem of object-oriented testing at different levels in general (e.g., [4] and [11-16]). Some testing techniques are specifically proposed to test object-oriented frameworks and their instantiations (e.g., [4-7, 17-27]).

Binder [4] suggests two different approaches for testing frameworks according to the availability of application-specific instantiations. The first approach, called *New Framework Test*, develops test cases for a framework that has few, if any, instantiations. The second approach, called *Popular Framework Test*, develops test cases for an enhanced version of a framework that has many application-specific instantiations. Tsai et al. [17] discuss the issues of testing instantiations developed with design patterns using object-oriented frameworks. The paper addresses testing from two viewpoints: that of framework developers, and that of instantiation designers. Framework developers test that the extensible patterns do allow the instantiation developer to extend the framework functionality. The instantiation designers should verify that the extension points are properly coded and tested. Wang et al. [18] propose providing the framework with reusable test cases that can be applied during the instantiation development stage. However, these test cases are limited to testing that the inherited framework features work correctly in the context of the instantiation classes that inherit them. Al Dallal [7, 26] and Al Dallal and Sorenson [19-22, 25] propose a technique to test the FICs at class level using reusable test cases built during the framework development stage. Al Dallal [23] proposes a technique to test the frameworks hook methods. The technique builds demo implementations for the hook methods and test suites to test the demo implementations. Kauppinen et al. [6] propose a criterion to evaluate the hook coverage of a test suite used to test hook methods. RITA [28] is a software tool that supports framework testing and automates the calculation of the hook method coverage measure. Al Dallal and Sorenson [24] propose a methodology to estimate the coverage of the cluster-based reusable test cases for framework instantiations. None of the techniques introduced for testing object-oriented frameworks and their instantiations considers retesting the framework at the application engineering stage.

Several techniques are introduced to test software product line and product family. In a software product line, variation points are points at which the products of a product family differ (i.e., each product has a different implementation, which is called a variant, for an abstract

© 2008 ACADEMY PUBLISHER

class associated with a variant point). Cohen et al. [29] suggest using combination testing strategies (e.g., [30]) to build test cases to test product line variants. In framework-based software product families, the variation points are the hook points, and implementations of the FICs are the variants. Tevanlinna et al. [1] identify four different strategies for modeling product family testing. The first strategy is to test product by product without considering the benefits of reuse (e.g., [31]). The second strategy tests the first product individually and the following products incrementally using regression testing techniques. The third strategy builds reusable test assets extensively during the domain engineering stage. These test assets are reused as-is or customized during the application engineering stage to test the product-specific aspects [8]. The fourth strategy applies unit testing in the domain engineering stage and integration, system, and acceptance testing at the application engineering stage. Only the first strategy considers retesting the assets each time an application is developed. However, in this strategy, the assets are retested from scratch and no reusability of testing previously applied is considered. The strategy introduced in this paper resembles the third product family testing strategy. The difference is that, in the third strategy, the reusable test assets are used to test the product-specific aspects, whereas, in our strategy, the reusable test assets are used to test the framework itself (i.e., the product core assets).

The testing areas for which reusability of test cases for object-oriented software are proposed and discussed include regression testing, testing subclasses, testing the use of class libraries, testing software product-lines, and testing object-oriented framework applications. An overview of research work addressing the last two areas is given above.

In regression testing [32-36], a modified version of the software is tested to provide confidence that the changed parts are behaving as intended and that the unchanged parts are not affected by the modifications in an unforeseen way. The test suite used to test the original version of the software or part of it is reused to test the modified version. In attempting to reuse the test suite or part of it, two problems have to be tackled: which test cases of the original test suite can or should be used to test the modified version, and which new test cases must be developed to test parts of the modified software [35]. In subclass testing [37], the superclass test suite or part of it must be reapplied in order to gain confidence that the inherited superclass features work correctly in the context of the subclass. In testing the use of the class libraries and the frameworks, Binder [4] states that the class library user and the framework user can reuse the class libraries test suite and the framework test suite, respectively, at the cluster-testing level without introducing new specific approaches.

The work introduced in this paper is different from those presented earlier in the following ways:

1. It determines the nontested framework part of the instantiation.
2. It provides test-case-reusing technique to test the framework part of the instantiation.
3. The introduced test-case-reusing technique selects test cases that test only the nontested part of the framework, reducing required testing time and effort.
4. The introduced test-case-reusing technique reuses the testing assets already built to test the framework during the domain engineering stage, reducing required testing time and effort.

III. The Framework Part Test-Case-Reusing Technique

In a former case study [7], we observed that more than half the instantiation classes are framework classes and the rest are either FICs or other instantiation classes. Therefore, when testing the instantiation, it is important to consider testing its framework part. The frameworks are already tested during their development process, but this testing is incomplete. Thus, it is useful to focus the testing on the framework part that is used in the instantiation but not covered during the framework testing stage. The test-case-reusing technique introduced in this paper consists of three steps, as follows:

Step 1: Identify the nontested hook methods

In TFTH technique, during the framework development stage, developers use CFGs to build different implementations of the hook methods through which the framework is tested. During the application engineering stage, developers implement the hook methods. The framework part of the instantiation must be tested using the hook methods implemented by the instantiation developers and not using the different implementations of the hook methods created using the CFG.

In TFTH technique, implementations of the hook methods are created from the CFG such that the CFG is branch covered. If the CFG contains a loop, the loop is iterated at most once. When the instantiation is developed, a loop in a CFG can be iterated more than once. Such a case is not used in testing the framework during the framework testing stage and must be used during the instantiation testing stage.

In addition, during the instantiation development stage, the developers can assign specific values for the parameters of the methods used in the hook methods or leave such parameters dynamic (i.e., the value can be assigned at run time). For example, when the *init* method is created using the *Initialize Account* hook given in Fig. 2, the instantiation developer can add branches to the system using the code statement

branch.addBranch(BranchName);

When this code statement is used, the instantiation developer can decide to assign a value to the *BranchName* parameter or leave the parameter dynamic. Most likely, the values assigned to the parameters are different than the test data applied during the framework testing stage. In this case, when testing the framework during the application engineering stage, the actual parameter values have to be considered.

© 2008 ACADEMY PUBLISHER

Finally, the framework is not tested during the framework domain engineering stage through extensible hook methods (i.e., open hook methods). This is because the implementation of such methods is left open for the instantiation developers. These methods are implemented during the application engineering stage and must be used to test the framework during the instantiation testing stage. As a result, the framework has to be tested during the application engineering stage through four types of hook methods:

(1) A hook method constructed by iterating more than once in a CFG loop.

(2) A hook method that includes parameters assigned to specific values during the application engineering stage.

(3) An open hook method.

(4) A hook method that includes a call for one of the above three types of hook methods

Such methods are marked *untested* and they have to be used in the test cases. Other methods are marked *tested* and they can be ignored when testing the framework during the application engineering stage unless there is a relation between such methods and the other methods marked *untested*.

Step 2: Remodel the round-trip path tree

The round-trip path tree is generated during the framework domain engineering stage using the HSTD. The HSTD includes transitions that are associated with hook method calls. Not all hook methods are implemented in each instantiation. The methods that are not implemented have to be ignored. In addition, some of the hook methods are marked *tested* in Step 1 and they can be ignored unless there is a relation between such methods and the other methods marked *untested*. The predicate of an HSTD transition can depend on a hook method not implemented in the instantiation. Such a transition has to be ignored; otherwise, reusing the corresponding test cases causes a compilation error indicating that the method is not defined. Finally, predicates of HSTD transitions can depend on variables assigned to specific values during the application engineering stage. If these values cause the predicate to evaluate to false, the transition cannot be executed and must be ignored.

Ignoring a transition in an HSTD causes ignoring of its corresponding transition in the round-trip path tree. When the round-trip path tree is constructed during the framework domain engineering stage, each transition in the HSTD is covered once. A state in the HSTD is covered a number of times equal to the number of its incoming transitions. As a result, a state in the HSTD can be represented by more than one corresponding state in the round-trip path tree. These corresponding states have the same identifier. For example, the *Frozen* state shown in Fig. 3 is represented three times in the round-trip path tree given in Fig. 5. Note that the *Frozen* state in Fig. 3 has three incoming transitions.

When a hook method is not implemented in the instantiation, transitions associated with it in the HSTD have to be deleted because they cannot be executed. Deleting a transition from the HSTD can result in a disjoint graph or unreachable states. For example, the developer of the banking framework instantiation can decide not to implement the *freeze* and *unfreeze* methods because the instantiation specifications do not require having frozen accounts. Deleting transitions associated with nonrequired methods from the HSTD causes a disjoint graph that includes the *Frozen* state and an associated transition. Deleting the corresponding transitions from the round-trip path tree does not cause any reachability problems. In this case, any resulting disjoint graphs and unreachable states can be deleted because they do not represent any necessary testable cases.

However, in some cases, deleting a transition from the HSTD does not result in creation of a disjoint graph or unreachable states. For example, in a banking framework instantiation, the developer chooses not to implement the transition originating from the *Open* state and ending at the *Inactive* state. This causes deletion of the transition from the HSTD given in Fig. 3. The *Inactive* state of the HSTD is still reachable through the *Frozen* state. However, deleting the corresponding transition and the resulting disjoint graphs from the round-trip path tree shown in Fig. 5 causes the outgoing transitions from the *Inactive* state to be unrepresented in the round-trip path tree. Some of these transitions can be associated with hook methods marked *untested* and these transitions must be used when testing the framework during the application engineering stage. To solve this problem, the outgoing transitions from a state that has been reached by a deleted transition must be reached by another path. In the round-trip path tree, this can be achieved by redirecting the transitions to be initiated from another state that has the same identifier as the destination state of the deleted transition. Fig. 6 illustrates the two deletion scenarios.

The first scenario occurs when a transition is deleted, but no disjoint graphs or unreachable states are created in the HSTD. For example, when transition A→B is deleted from the HSTD given in Fig. 6(i), the resulting HSTD remains a connected graph in which all states are reachable. Therefore, all remaining states and transitions represented in the resulting HSTD have to be represented in the remodeled round-trip path tree. Deleting transition A→B from the corresponding round-trip path tree given in Fig. 6(ii) creates a disjoint graph consisting of states B and D and the transition between them. This disjoint graph must be reconnected to the round-trip path tree by redirecting the transition to be initiated from state B, as shown in Fig. 6(iii). Note that the remodeled-round-trip path tree represents all the states and transitions contained in the remodeled HSTD. The second scenario occurs when a disjoint graph or unreachable state is created. For example, when transition A→C is deleted from the HSTD given in Fig. 6(i), the resulting HSTD contains state C, which becomes unreachable. In this case, the disjoint graphs or the unreachable states have to be deleted from the HSTD with the transitions connected to them. Deleting corresponding states and transitions from the round-trip path tree does not require any extension for an existing path. For example, when transition A→C is

© 2008 ACADEMY PUBLISHER

deleted from the round-trip path tree, the remodeled tree is the result of deleting transition A→C and the resulting disjoint graph from the original round-trip path tree.

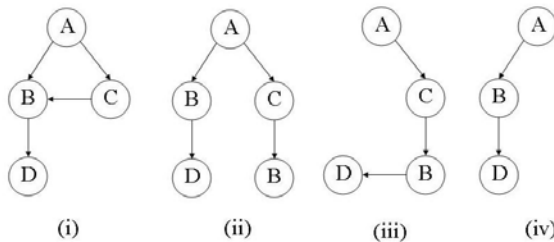

Figure 6. Transition deletion mechanism. (i) HSTD, (ii) round-trip path tree constructed from the HSTD, (iii) round-trip path tree after deleting the transition between A and B states, (iv) round-trip path tree after deleting the transition between A and C states.

In summary, when deleting a transition from the HSTD, it is not required to reproduce the round-trip path tree. The round-trip path tree can be remodeled without referring to the HSTD. In this case, if deleting a transition from the round-trip path tree results in a disjoint graph, the disjoint graph is reconnected to the tree, as illustrated in Step 3.2.2 in the procedure given in Fig. 7.

Similar actions have to be taken when the values assigned to the parameters used in the transition predicate cause the transition not to be executed.

The round-trip path tree transitions represented by dotted links are used to consider the CFG of the hook methods called implicitly. Since the CFGs are no longer used during the instantiation testing stage, such transitions are deleted. When deleting a dotted transition, the transition source state must be deleted and the incoming transition to the source state has to be redirected to end at the destination state of the dotted transition. For example, the transition from *initializing* state to *Open* state in Fig. 5 is dotted. Therefore, the transition and the *initializing* state must be deleted, and transition labeled *1* must be connected to *Open* state.

After performing the deletion process mentioned above, all remaining transitions in the remodeled round-trip path tree are executable. However, it is possible that the remodeled round-trip path tree includes paths that are identical, in terms of sequence of hook method calls and transition predicates, to the paths in the original round-trip path tree. In addition, it is possible that the implementations of all hook methods invoked when covering such paths during the domain engineering stage are identical to the implementations of the corresponding hook methods coded during the application engineering stage. Such paths in the remodeled round-trip path tree can be ignored and not reapplied at the application engineering stage because they cover cases already tested at the domain engineering stage. According to the marking schema used in this paper, this case occurs when a path in the remodeled round-trip path tree is identical to a path in the original round-trip path tree in terms of sequence of hook method calls and transition predicates and all hook methods included in the sequence are marked *tested*.

The procedure given in Fig. 7 remodels the round-trip path tree used during the framework development stage according to the above discussion. The resulting tree includes transitions that are associated with hook methods marked *untested* or *tested* but are necessary to cover transitions associated with *untested* hook methods.

Input: A round-trip path tree including transitions associated with marked hook methods
Output: A remodeled round-trip path tree that can be used for testing the framework during the application engineering stage.
Procedure:
1. For each dotted transition *t* from state *s* to state *d* do
   1.1. delete transition *t*.
   1.2. redirect the incoming transition to state s to end at state *d*.
   1.3. delete state *s*.
2. Mark all transitions *not visited*.
3. *while* not all transitions are marked *visited do*
   3.1. Pick a closest *not visited* transition *t* to a leaf state.
   3.2. *if* (event associated with transition *t* represents a call for a hook method not implemented in the instantiation) *OR* (the supplied values of the parameters are not sufficient to make the predicate of transition *t* TRUE) *then*
      3.2.1. delete transition *t*.
      3.2.2. *for* each transition *tg* outgoing from the destination state of transition *t*
         Connect_Sub_Tree(transition *tg*)
   3.3. *else*
      3.3.1. delete from transition *t* any predicate that always evaluates to TRUE.
      3.3.2. mark transition *t visited*.
4. Mark all transitions *not visited*.
5. *while* not all transitions connected directly to leaf states are marked *visited do*
   5.1. Pick a transition *t* connected to a leaf state s.
   5.2. *if* (all events associated to transitions from the root state to state s represent calls to hook methods marked *tested* AND all transitions from the root state to state s do not have predicates including variables assigned to specific values at application engineering stage) *then*
      5.2.1. delete transition *t* and its destination state.
   5.3. *else*
      5.3.1. mark transition *t visited*.

Connect_Sub_Tree(transition *t*)
   if a state *s* in the round-trip path tree has the same identifier as the identifier of the source state of transition *t* then
      Redirect transition *t* to be initiated from state *s*
   else
      Delete transition *t*
      for each transition *tg* outgoing from the destination state of transition *t*
         Connect_Sub_Tree(transition *tg*)

Figure 7. Re-modeling round-trip path tree procedure

Suppose that the banking framework is used to implement an instantiation to be used in managing bank accounts. The bank has a policy to inactivate any account with no activity for five years (i.e., 1825 days). Such accounts cannot be settled unless they are reactivated. Finally, the bank has three branches. The given specification causes the *settle* method not to be implemented; therefore, its corresponding transition is deleted from the round-trip path tree. The CFG of the *init* method includes a loop iterated more than once to create objects for the three branches. The *NewAccount* method calls the *init* method, and therefore, it is marked *untested* and the corresponding transition is kept in the re-modeled

© 2008 ACADEMY PUBLISHER

round-trip path tree. Since all paths in the round-trip path tree include the transition associated with a call for *NewAccount* hook method, the paths are kept in the remodeled round-trip path tree.

In another similar scenario, if the bank has one branch, the CFG of the *init* method does not include a loop iterated more than once. Therefore, the hook method associated with the transition labeled 1 in Fig. 5 is marked *tested*. All other methods are also marked *tested* because none of them is of one of the four types of hook methods mentioned in Section III (Step 1). When the procedure given in Fig. 7 is applied, Transition 2 is deleted, in Step 1, because it is dotted. In Step 2, all left transitions are marked *not visited*. In Step 3, none of the transitions is deleted because all of them are associated to calls of hook methods implemented in the scenario. In Step 4, all left transitions are marked *not visited* again. In Step 5, transitions 3, 4, 5, 6, 7, 8, 9, 11, 15, and 16 are deleted because the conditions stated in Step 5.2 are satisfied. Transitions 12, 13, and 17 are kept because they contain predicates that include variables assigned to certain values for the number of days to inactivate an account. Transitions 1, 10, and 14 are kept because they are included in paths containing kept transitions (i.e., Transitions 12, 13, and 17). The remodeled round-trip path tree is shown in Fig. 8.

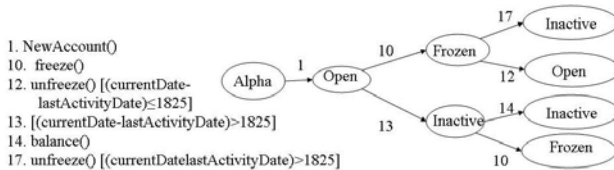

Figure 8. Re-modeled round-trip path tree example

Step 3: Produce round-trip path test cases.

Each test case exercises a single round-trip path in the remodeled round-trip path tree. The round-trip path starts at *alpha* state of the remodeled round-trip path tree and ends at one of the leaf states of the tree. In this step, it is not required to create any test case from scratch. In the remodeled tree, the paths that are identical to the corresponding paths in the original tree are covered using the same test cases used to cover the original tree paths during the framework development stage. If the path in the remodeled tree contains a predicate that has variables assigned to certain values, the corresponding test case has to be modified to include the provided values. Fig. 9 shows a Java coded test case that covers the path *Alpha→Open→Frozen→Open* in the remodeled tree given in Fig. 8. In the test case, the modified parts are bolded.

In the remodeled tree, a path can be extended from a path in the original round-trip path tree. In this case, the corresponding test case has to be augmented in order to consider the predicates and to include calls for the methods associated with the added transitions.

```
public class TEST3_NewAccount{
  public TEST3_NewAccount(){
   /* Test transition: source state: Alpha,
      sink state: Open, event:
      NewAccount(amount),
      predicates:amount>=0 */
   float amount=1;
   NewAccount o = new NewAccount(amount);
   /** @assert((o.balance()>=0)
      &&((o.getCurrentDate()-
      o.getLastActivityDate())<o.getMaxPeriod
      ()) && !(o.isFrozen()))*/
   /* Test transition: source state: Open,
      sink state: Frozen, event: freeze(),
      predicates: none */
   o.freeze();
   /** @assert((o.balance()>=0) &&
      ((o.getCurrentDate()-
      o.getLastActivityDate())<o.getMaxPeriod
      ()) && (o.isFrozen()))*/
   /* Test transition: source state: Frozen,
      sink state: Open, event: unfreeze(),
      predicates: ((o.getCurrentDate()-
      o.getLastActivityDate())<=1825) */
   o.unfreeze();
   /** @assert((o.balance()>=0) &&
      ((o.getCurrentDate()-
      o.getLastActivityDate())<=1825) &&
      !(o.isFrozen()))*/
  }
}
```

Figure 9. The Java implementation of a modified test case

In Step 2, two example scenarios are mentioned. In the first one, the remodeled round-trip path tree is found to be the same as the original one except for deletion of transition labeled 15. As a result, all the test cases built during the domain engineering stage are reused as-is during the application engineering stage except the test case including transition 15. This test case has to be modified to discard the call for *settle* hook method. In the second example scenario for which the round-trip path tree is given in Fig. 8, one of the test cases built during the domain engineering stage is not applicable because *settle* hook method is not implemented during the application engineering stage. In addition, four test cases are applied as-is to test the framework at the application engineering stage. None of the other test cases applied during the domain engineering stage have to be reapplied during the application engineering stage because they cover cases already tested during the domain engineering stage.

IV. Performance Evaluation

In this section, we evaluate the proposed test-case-reusing technique by applying it to test the framework portion of two WaveFront framework instantiations. The results are used to show the adequacy of the test-case-reusing technique in terms of reducing the testing time and effort.

A. The Used Framework and Instantiations

WaveFront Pattern (WFP) [38] is a pattern that supports the computation of dependent elements. The

© 2008 ACADEMY PUBLISHER

pattern is used to generate frameworks automatically using CO_2P_3S parallel programming system [39]. A generated WaveFront Pattern framework was considered in this experiment. The framework was relatively small, consisting of six classes and about 150 lines of code. Three hook descriptions were used to document how to use the framework. In this experiment, the TFTH technique was applied to build the test cases to test the framework at the domain engineering stage. Two FICs were identified and their HSTDs were constructed. The hook descriptions were followed to construct the CFGs of the hook methods. Eight CFGs were constructed. The CFGs were traversed to construct the different implementations of the hook methods. The two round-trip path trees for the two FICs were built using the HSTDs and traversed to construct the test cases. The two round-trip path trees included 22 paths. Each path was used to build a test case. Each test case exercising a hook method that has different implementations created using the CFG exercises each implementation at least once.

Two instantiations built using the WaveFront framework were considered in this experiment. The first instantiation was called Skyline Matrix and consisted of about 450 lines of code and 12 classes. Six of the classes were framework classes and the rest were developed by the instantiation developer. The second instantiation was called Matrix Block, consisting of about 330 lines of code and 13 classes. Six of the classes were framework classes and the rest were developed by the instantiation developer.

B. *Applying the Test-Case-Reusing Technique*

The test-case-reusing technique was applied to determine the reusable test cases for retesting the framework. The purpose of the experiment was to demonstrate the usefulness of the proposed technique in (1) reusing the test cases built during the domain engineering stage to test the framework during the application engineering stage and (2) neglecting test cases already applied during the domain engineering stage. The implemented FICs in the two instantiations were manually traced to identify the nonimplemented and the nontested hook methods according to the discussion provided in Section III. It is important to note that this manual tracing would not be required if the details of the hook enactment process are reported during the implementation process of the FICs. For each instantiation, the experiment was conducted twice. The first time, the introduced technique was used without considering the marking schema (i.e., *tested* or *untested*) of the hook methods. Only the transitions associated with nonimplemented hook methods or nonexecutable predicates were deleted from the original round-trip path tree. This means that only the first three steps of the procedure given in Fig. 7 were applied. The second time, the introduced technique was used as-is (i.e., all the steps of the procedure given in Fig. 7 were applied). The purpose of conducting the experiments twice was to demonstrate the usefulness of the marking schema of the hook methods in ignoring test cases already covered during the domain engineering stage and consequently reducing the number of applied test cases. Ignoring a test case already covered during the domain engineering stage implies reducing the time and effort required for testing the framework during the application engineering stage.

In each experiment, the number of reused test cases as-is, the number of modified test cases, the number of ignored test cases, and the total number of required test cases were counted and reported in the third, fourth, fifth, and sixth rows of Table 1, respectively. The first and second rows of the table report the name of the framework instantiation and the experiment identifier, respectively. The second, third, fourth, and fifth columns report the results of applying the first and second experiments on the Skyline Matrix and Matrix Block instantiations, respectively.

Table 1. Results of applying the proposed test-case-reusing technique to test the framework part of the WaveFront Pattern framework instantiations.

Instantiation Name	Skyline Matrix		Matrix Block	
Experiment	1st Experiment	2nd Experiment	1st Experiment	2nd Experiment
Number of reusable test cases as-is	16	1	11	1
Number of modified test cases	6	6	7	7
Number of ignored test cases	0	15	0	10
Total number of test cases applied during the application engineering stage	22	7	18	8

In the considered instantiations, the implemented FICs did not contain methods not introduced in the hook descriptions. As a result, none of the test cases required to test the framework during the application engineering stage was created from scratch. Despite the fact that this is not always true, our experience in studying FICs gives an indication that usually a relatively small percentage of FIC methods are new. This experience gives an indication that applying the introduced technique greatly reduces framework testing efforts during the application engineering stage. This is because most or all test cases required for testing the framework during the application engineering stage are reused versions of the test cases built during the domain engineering stage. In the conducted experiments, on average, about 60% of the required framework test cases during the application engineering stage were ignored because they covered cases already tested during the domain engineering stage. This gives an indication that the marking schema used in the introduced technique for the hook methods is useful in ignoring test cases already covered during the domain engineering stage. Consequently, this gives an indication that the introduced technique is effective in reducing testing time and effort.

© 2008 ACADEMY PUBLISHER

C. Time Efficiency Assessment for the Test-Case-Reusing Technique

To demonstrate the efficiency of the introduced test-case-reusing technique in terms of reducing the time and effort required for retesting the framework, five graduate students at Kuwait University conducted a comparison experiment individually. The experiment was performed in two stages. In the first stage, the students were asked to perform testing for the WaveFront framework used in the two considered applications from scratch (i.e., without considering the test-case-reusing technique introduced in this paper). In the second stage, the students were asked to apply the test-case-reusing technique to perform the required testing. The time required for each experiment stage was reported and compared.

In the first stage, the students were taught how to build a state transition model for a class using the specifications given in the hook descriptions. The students were given the hook descriptions of the WaveFront framework and the implemented hook method details. The students took an average of about 12 hours and 46 minutes to draw the state transition models for the four FICs included in the two applications. Unfortunately, none of the models was complete because the algorithm used to construct the models from the method specifications [20] is difficult to be applied manually. The students were taught how to produce a round-trip path tree from a state transition model and were given the revised models with the invariants of the states included in the models. The average time used to produce all required round-trip path trees was about 1 hour and 12 minutes. Finally, the students were taught how to write test drivers in Java for the round-trip paths included in the trees. A total of 40 test drives were produced from all considered classes. The average time spent in writing, compiling, and executing the required test drivers was 12 hours and 32 minutes. In summary, the students were taught several skills and techniques for producing the required test drivers. The students were supplied with the framework hook descriptions and the implemented hook method details. In addition, the students were assisted in completing the models and provided with the invariants of the states to enable them to write the test drivers. Each student took an average of 26 hours and 30 minutes to produce the required test cases from scratch.

In the second stage, the students were given the implemented hook method details and the 22 test drivers applied during the framework domain engineering stage. The students were taught first how to use the implemented hook method details to list the names of the methods implemented in the application and to list the names of the methods marked *untested*. The average time spent constructing the two lists for all the considered methods was 11 minutes. After that, the students were taught how to modify the given test drivers according to the first three steps of the procedure given in Fig. 7. The students spent an average of 14 minutes modifying the 22 test drivers for the two considered applications. In the second application, four of the test drivers were found to be inapplicable because they include a method not implemented in the application. These test drivers were ignored. The modifications applied to the rest of the test drivers were similar to the ones applied to the test drivers of the first application. The students were then taught how to apply the last two steps of the procedure given in Fig. 7 to neglect unnecessary test drivers. The students spent an average of 7 minutes in detecting the unnecessary test drivers. Finally, the students spent an average of 17 minutes to compile and execute the test drivers. In summary, the students were taught how to apply the test-case-reusing technique and were supported with the framework test drivers and a file containing the implemented hook method details. The students spent an average of 49 minutes to identify, modify, compile, and execute the reusable test drivers.

Clearly, many fewer skills needed to be taught in the second stage of the experiment. The students in the second stage of the experiment were able to complete the entire testing process without major assists, whereas major revision was required in the first stage of the experiment for the testing models. In addition, the amount of information given in the second stage of the experiment was less than that given in the first stage of the experiment. Finally, the average time spent in the second stage of the experiment was 3.08% of the average time spent in the first stage of the experiment. The experiment shows that applying the test-case-reusing technique introduced in this paper reduces required testing time and effort greatly. The results of the experiment are summarized in Table 2.

Table 2. Time efficiency results of retesting the WaveFront Pattern framework with and without applying the proposed test-case-reusing technique.

Retesting the framework without applying the test-case-reusing technique		Retesting the framework by applying the test-case-reusing technique	
Tasks	Average time (minutes)	Tasks	Average time (minutes)
1. Building state transition models.	766	1. Identifying the implemented and untested hook methods.	11
2. Producing round-trip path trees.	72	2. Modifying the framework test drivers.	14
3. Writing, compiling, and executing test drivers.	752	3. Detecting unnecessary test drivers.	7
		4. Compiling and executing the reusable test drivers.	17
Total	1590	Total	49 (3.08%)

V. AUTOMATION

Generally, software testing is a time-consuming and costly task. Therefore, automation is a vital issue in software testing. The techniques introduced in this paper for identifying cases that have to be retested and for reusing framework test cases already available are systematic tasks and can be automated. To demonstrate

© 2008 ACADEMY PUBLISHER

the practicality of the introduced technique, a supporting tool called JFramework Re-Tester has been developed. The tool uses outputs of two other tools. The first one is JFramework Tester [5], a tool that supports the testing of object-oriented frameworks written in Java. JFramework Tester automates TFTH testing, produces framework test cases, and stores these test cases in the framework database. The second tool is Hook Master [10], a tool that enacts hook description statements and builds corresponding Java implementation for hook methods during the application engineering stage. In this research, the Hook Master tool is extended to produce a file that reports data required to identify whether the implemented hook method must be considered in retesting the framework. For each hook method created using the Hook Master tool, the file reports (1) the hook method name, (2) whether the hook description used to build the hook method includes construction loops iterated more than once during the hook method implementation process, (3) whether the hook description used to build the hook method includes parameters assigned to specific values during the hook method implementation process, (4) whether the hook method is an open type, and (5) if any, the names of the other hook methods called by the hook method under consideration.

JFramework Re-Tester tool is implemented in Java and contains 8 classes and about 3250 lines of executable code. The high-level description of the tool is shown in Fig. 10. The user of the tool, the framework application developer, selects the framework to be retested using a browser. The framework database includes the framework code, the framework hook descriptions, the framework test cases produced by JFramework Tester tool, and the implemented hook method data files produced by the extended Hook Master tool. The *Test Case Parser* module of the JFramework Re-Tester tool parses the framework test cases and stores them in objects organized in a link list data structure. Simultaneously, the *Implemented Hook Methods Parser* module of the tool parses the implemented hook method data and stores them in objects organized in a link list data structure. The parsed implemented hook method data are used by the *Nontested Hook Methods Detector* module of the tool to identify the nontested hook methods according to the criteria discussed in Section III (Step 1). The parsed implemented hook method data are also used with the parsed test cases by the Test Case Modifier module of the tool to modify the test cases according to the first three steps of the procedure given in Fig. 7. The modified test cases and the names of the hook methods marked untested are used by the Applicable Test Case Detector module of the tool to decide on the test cases that have to be applied to retest the framework during the application engineering stage.

The applicable test cases are stored in the framework database and corresponding Java code is generated by the Test Drivers Builder module of the tool. The Test Drivers Builder module instruments the test drivers (i.e., implementations of the test cases) by the state invariants written in DbC language [40] and stored in the framework database. The *Test Drivers Executer* module of the tool compiles the test drivers and the implemented FICs using the dbc_javac compiler of the Jcontract tool [41]. The Jcontract compiler checks the DbC specifications in the Javadoc comments, generates instrumented .java files with extra code to check the contracts (i.e., preconditions and postconditions) in the Javadoc comments, and compiles the instrumented .java files with the `javac` compiler. The resulting .class files are instrumented with extra bytecodes to check the contracts at runtime. Finally, the *Test Drivers Executer* module executes the test drivers and uses *Jcontract* tool to automatically check the contracts at runtime and report any violations found. The appendix shows a complete example that uses the JFramework Re-Tester tool during the application engineering stage to retest the banking framework when the *NewAccount* FIC class is implemented.

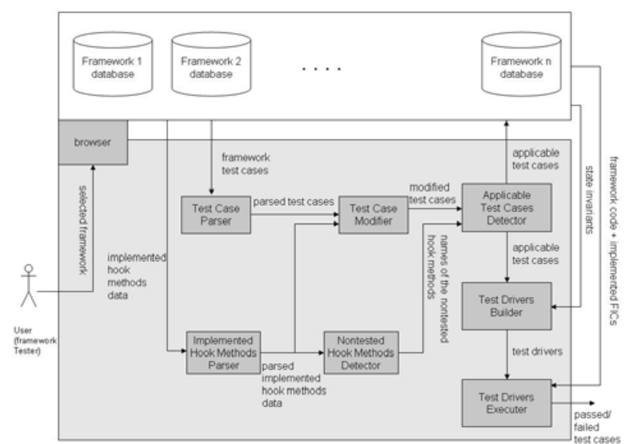

Figure 10. The high level design of the JFramework Re-Tester tool

VI. CONCLUSIONS AND FUTURE WORK

This paper introduces a test-case-reusing technique to reuse the framework test suite already applied during the domain engineering stage to test the framework during the application engineering stage. The test-case-reusing technique uses the same framework testable models proposed in the TFTH technique. The test-case-reusing technique first identifies the nontested portion of the framework. Then, it remodels the round-trip path tree used during the framework domain engineering stage to eliminate the inclusion of the nonimplemented hook methods and to ignore unnecessary tested hook methods. Finally, the technique identifies the framework test cases that can be reused as-is or augmented. As a result, the introduced test-case-reusing technique for the framework part of the instantiation makes use of the work performed during the framework domain engineering stage in several forms, including (1) reusing the framework testing models, (2) ignoring the tested part of the framework during the framework development stage, and (3) reusing, augmenting, or modifying the framework test cases.

Two experiments were conducted to show the adequacy of the proposed test-case-reusing technique in

terms of reducing testing time and effort. The experiments show that the test-case-reusing technique effectively reuses framework test cases to retest the framework during the application engineering stage. In addition, the experiments show that the test-case-reusing technique greatly reduces the time and effort required for retesting the framework during the application engineering stage. The test cases considered in the test-case-reusing technique are produced using the round-trip path coverage technique. The effectiveness of the round-trip path coverage test cases in terms of error detection power is studied generally in [42] and specifically in a framework context in [5].

A supporting tool that automates the complete retesting process of the framework during the application engineering stage using the introduced techniques is developed. The tool is limited for testing frameworks developed in Java.

The introduced technique and tool assume that the framework under testing is provided with hook descriptions. However, this might not be the case for most of the available frameworks; typically frameworks are provided with steps to illustrate how to use the framework. These steps can be formed in hook descriptions instead of in the informal description. Such a task requires a clear understanding of hook notation and hook description formal language. We have exercised this task with several frameworks and found it to be straightforward. Hooks were originally introduced as an aid to show where and how to extend object-oriented frameworks in constructing complete software applications. This paper shows that the hooks can also be used as an aid in retesting the framework. We expect that extending the benefits and uses of the hooks will encourage framework developers to show considerable interest in documenting their frameworks using hooks.

In the area of future work, the proposed technique for reusing the test cases does not guarantee that the modified round-trip path tree is free of infeasible paths. Infeasible paths are ones that cannot be executed because of conflicting or impossible to satisfy predicates. To solve this problem, we have to either detect the infeasible paths and avoid using them in generating the test drivers [43] or ignore any test driver that has violated preconditions [40]. Detection of the infeasible paths is not implemented in the developed supporting tool. In addition, the tool does not automate conversion of predicates into Java code statements. The corresponding code statements are associated with the predicates in the file that includes the framework test cases.

APPENDIX A

The following is the content of the test cases generated using JFramework Tester tool. This file is an input file to the JFramework Re-Tester tool. Each test case includes the description of the sequence of transitions exercised in the test case. Each transition includes the hook method name, hook method parameters, predicates with the corresponding Java code, actions, and source and destination states.

```
Class:NewAccount
TC 1
NewAccount()["float
amount=1;"],from:Alpha,to:Open
balance(),from:Open,to:Open
TC 2
NewAccount()["float
amount=1;"],from:Alpha,to:Open
deposit(amount)["amount=1;"]/balance=balanc
e+amount,from:Open,to:Open
TC 3
NewAccount()["float
amount=1;"],from:Alpha,to:Open
withdraw(amount)[(balance()-
amount)>=0"amount=o.balance();"]/balance=ba
lance-amount,from:Open,to:Open
TC 4
NewAccount()["float
amount=1;"],from:Alpha,to:Open
withdraw(amount)[(balance()-
amount)<0"amount=1+o.balance();"]/balance=b
alance-amount,from:Open,to:Overdrawn
deposit(amount)[(amount+balance())<0"amount
=o.balance();"]/balance=balance+amount,from
: Overdrawn,to:Overdrawn
TC 5
NewAccount()["float
amount=1;"],from:Alpha,to:Open
withdraw(amount)[(balance()-
amount)<0"amount=1+o.balance();"]/balance=b
alance-amount,from:Open,to:Overdrawn
balance(),from:Overdrawn,to:overdrawn
TC 6
NewAccount()["float
amount=1;"],from:Alpha,to:Open
withdraw(amount)[(balance()-
amount)<0"amount=1+o.balance();"]/balance=b
alance-amount,from:Open,to:Overdrawn
deposit(amount)[(amount+balance())>=0"amoun
t=1-o.balance();"]/balance=balance+amount,
from:Overdrawn,to:open
TC 7
NewAccount()["float
amount=1;"],from:Alpha,to:Open
freeze(),from:Open,to:Frozen
balance(),from:Frozen,to:Frozen
TC 8
NewAccount()["float
amount=1;"],from:Alpha,to:Open
freeze(),from:Open,to:Frozen
unfreeze()[(currentDate-
lastActivityDate)>maxPeriod"o.setLastActivi
tyDate(o.getCurrentDate()-
o.getMaxPeriod());"],from:Frozen,to:Inactiv
e
TC 9
NewAccount()["float
amount=1;"],from:Alpha,to:Open
freeze(),from:Open,to:Frozen
unfreeze()[(currentDate-
lastActivityDate)<=maxPeriod],from:Frozen,t
o:Open
TC 10
NewAccount()["float
amount=1;"],from:Alpha,to:Open
[(currentDate-
lastActivityDate)>maxPeriod"o.setLastActivi
```

© 2008 ACADEMY PUBLISHER

```
tyDate(o.getCurrentDate()-
o.getMaxPeriod());"],from:Open,to:Inactive
balance(),from:Inactive,to:Inactive
TC 11
NewAccount()["float
amount=1;"],from:Alpha,to:Open
[(currentDate-
lastActivityDate)>maxPeriod"o.setLastActivi
tyDate(o.getCurrentDate()-
o.getMaxPeriod());"],from:Open,to:Inactive
freeze(),from:Inactive,to:Frozen
TC 12
NewAccount()["float
amount=1;"],from:Alpha,to:Open
[(currentDate-
lastActivityDate)>maxPeriod"o.setLastActivi
tyDate(o.getCurrentDate()-
o.getMaxPeriod());"],from:Open,to:Inactive
settle(),from:Inactive,to:Omega
TC 13
NewAccount()["float
amount=1;"],from:Alpha,to:Open
close(),from:Open,to:Omega
```

APPENDIX B

The following is the content of the implemented hook method details generated by the extended Hook Master tool. This file is an input file to the JFramework Re-Tester tool. The file lists the names of the implemented hook methods, whether the hook description used to build the hook method includes construction loops iterated more than once during the hook method implementation process, whether the hook description used to build the hook method includes parameters assigned to specific values during the hook method implementation process, whether the hook method is an open type, and if any, the names of the other hook methods called by the hook method under consideration.

```
Class:NewAccount
Hook method:NewAccount
Number_of_iterations:1
Hook method:balance
Hook method:deposit
Hook method:withdraw
Hook method:freeze
Open hook
Hook method:unfreeze
Open hook
Hook method:close
```

APPENDIX C

The following are the test drivers generated using the JFramework Re-Tester tool. The tool found that TC 13 contained in the file given in Appendix A includes a call to *settle* hook method, which is not included in the file given in Appendix B. Therefore, TC 12 is not reused. Hook methods *freeze* and *unfreeze* are declared in the file contained in Appendix B as open hook methods. Therefore, they are marked untested and any test case that includes a call for one of them has to be re-exercised. This results in reusing test cases 7, 8, 9, and 11 as-is. None of the other test cases included in the file in Appendix A exercises cases not already tested during the framework domain engineering stage. Therefore, these test cases are ignored.

```
public class TEST7_NewAccount{
    public TEST7_NewAccount(){
        /* Test transition: source state:
          Alpha, sink state: Open, event:
          NewAccount(amount), predicates:
          amount>=0 */
        float amount=1;
        NewAccount o = new
          NewAccount(amount);

        /** @assert((o.balance()>=0) &&
          ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMax
          Period()) && !(o.isFrozen())) */
        /* Test transition: source state:
          Open, sink state: Frozen, event:
          freeze(), predicates: none */
        o.freeze();

        /** @assert((o.balance()>=0) &&
          (o.isFrozen())) */
        /* Test transition: source state:
          Frozen, sink state: Frozen,
          event: balance(), predicates:
          none */
        o.balance();

        /** @assert((o.balance()>=0) &&
          (o.isFrozen())) */
    }
}

public class TEST8_NewAccount{
    public TEST8_NewAccount(){
        /* Test transition: source state:
          Alpha, sink state: Open, event:
          NewAccount(amount), predicates:
          amount>=0 */
        float amount=1;
        NewAccount o = new
          NewAccount(amount);

        /** @assert((o.balance()>=0) &&
          ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMax
          Period()) && !(o.isFrozen())) */
        /* Test transition: source state:
          Open, sink state: Frozen, event:
          freeze(), predicates: none */
        o.freeze();

        /** @assert((o.balance()>=0) &&
          (o.isFrozen())) */
        /* Test transition: source state:
          Frozen, sink state: Inactive,
          event: none, predicates:
          (getCurrentDate() -getLastAct
          ivityDate())>=getMaxPeriod() */
        o.setLastActivityDate(o.get
          CurrentDate()-o.getMaxPeriod());

        /** @assert((o.balance()>=0) &&
          ((o.getCurrentDate()-
          o.getLastActivityDate())>=o.getMa
          xPeriod()) && !(o.isFrozen())) */
```

© 2008 ACADEMY PUBLISHER

```
    }
}

public class TEST9_NewAccount{
    public TEST9_NewAccount(){
        /* Test transition: source state:
          Alpha, sink state: Open, event:
          NewAccount(amount), predicates:
          amount>=0 */
        float amount=1;
        NewAccount o = new
          NewAccount(amount);

        /** @assert((o.balance()>=0) &&
          ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMax
          Period()) && !(o.isFrozen())) */
        /* Test transition: source state:
          Open, sink state: Frozen, event:
          freeze(), predicates: none */
        o.freeze();

        /** @assert((o.balance()>=0) &&
          (o.isFrozen())) */
        /* Test transition: source state:
          Frozen, sink state: Open, event:
          unfreeze(), predicates:
          balance>=0 */
        /* The following is a predicate
          assertion */
        /** @assert(o.balance()>=0) */
        o.unfreeze();

        /** @assert((o.balance()>=0) &&
          ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMax
          Period()) && !(o.isFrozen())) */
    }
}

public class TEST11_NewAccount{
    public TEST11_NewAccount(){
        /* Test transition: source state:
          Alpha, sink state: Open, event:
          NewAccount(amount), predicates:
          amount>=0 */
        float amount=1;
        NewAccount o = new
          NewAccount(amount);

        /** @assert((o.balance()>=0) &&
          ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMax
          Period()) && !(o.isFrozen())) */
        /* Test transition: source state:
          Open, sink state: Inactive,
          event: none, predicates:
          (getCurrentDate() -
          getLastActivityDate())>=getMaxPer
          iod() */
           o.setLastActivityDate(o.getCurre
          ntDate()-o.getMaxPeriod());

        /** @assert((o.balance()>=0) &&
          ((o.getCurrentDate()-
          o.getLastActivityDate())>=o.getMa
          xPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state:
          Inactive, sink state: Frozen,
          event: freeze(), predicates: none
          */
        o.freeze();

        /** @assert((o.balance()>=0) &&
          (o.isFrozen())) */
    }
}

  public class DRIVER_MyAccount{
       public static void main(String
           args[]){
           //invoking reusable test drivers
           new TEST7_NewAccount();
           new TEST8_NewAccount();
           new TEST9_NewAccount();
           new TEST11_NewAccount();
      }
  }
```

ACKNOWLEDGMENT

The authors would like to acknowledge the support of this work by Kuwait University Research Grant WI01/06.

REFERENCES

[1] A. Tevanlinna, J. Taina, and R. Kauppinen, "Product Family Testing: a Survey," ACM SIGSOFT Software Engineering Notes, 2004, Vol. 29, No. 2, pp.12-18.

[2] J. Bosch, Design and Use of Software Architectures, Addison-Wesley, 2000.

[3] K. Beck and R. Johnson, "Patterns Generated Architectures," Proc. of ECOOP 94, 1994, pp. 139-149.

[4] R. Binder. Testing Object-Oriented Systems, Addison Wesley, 1999.

[5] J. Al Dallal and P. Sorenson, "System Testing for Object-Oriented Frameworks Using Hook Technology," Proc. of the 17th IEEE International Conference on Automated Software Applications (ASE'02), Edinburgh, UK, September 2002, pp. 231-236.

[6] R. Kauppinen, J. Taina, and A. Tevanlinna, "Hook and Template Coverage Criteria for Testing Framework-Based Software Product Families," In Proceedings of the International Workshop on Software Product Line Testing, Boston, Massachusetts, USA, 2004.

[7] J. Al Dallal, "Class-Based Testing of Object-Oriented Framework Interface Classes," Ph.D. Thesis, University of Alberta, Department of Computing Science, 2003.

[8] J. McGregor, "Testing a Software Product Line," Technical Report CMU/SEI-2001-TR-022, Software Engineering Institute, Pittsburgh, PA, 2001.

[9] 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, 1997, pp. 491-501.

[10] G. Froehlich, "Hooks: an Aid to the Reuse of Object-Oriented Frameworks," Ph.D. Thesis, University of Alberta, Department of Computing Science, 2002.

[11] D.A. Sykes and J.D. McGregor, Practical Guide to Testing Object-Oriented Software, Addison Wesley, 2001.

[12] N. Daley, D. Hoffman, and P. Strooper, "A Framework for Table Driven Testing of Java Classes," Software-Practice and Experience, 2002, 32, pp. 465-493.

© 2008 ACADEMY PUBLISHER

[13] S. Mouchawrab, L. C.Briand, and Y. Labiche, "A Measurement Framework for Object-Oriented Software Testability", Journal of Information and Software Technology," 2005, Vol. 47, No. 15, pp. 979-997.
[14] L. C. Briand, Y. Labiche, and M. Sówka, "Automated, Contract-Based User Testing of Commercial-off-the-Shelf Components," Proceedings of the 28th International Conference on Software Engineering (ICSE), Shanghai, China, 2006.
[15] L. Gallagher and J. Offutt, "Automatically Testing Interacting Software Components," Workshop on Automation of Software Test (AST 2006), Shanghai, China, 2006.
[16] L. Gallagher, J. Offutt, and A. Cincotta, "Integration Testing of Object-Oriented Components Using Finite State Machines," Journal of Software Testing, Verification and Reliability, 2007, Vol. 17, No. 1, pp. 215-266,
[17] W. Tsai, Y. Tu, W. Shao, and E. Ebner. "Testing Extensible Design Patterns in Object-Oriented Frameworks Through Scenario Templates," 23rd Annual International Computer Software and Applications Conference, Phoenix, Arizona, 1999.
[18] Y. Wang, D. Patel, G. King, I. Court, G. Staples, M. Ross, and M. Fayad, "On Built-in Test Reuse in Object-Oriented Framework Design," ACM Computing Surveys (CSUR), 2000, Vol. 32(1es), pp. 7-12.
[19] J. Al Dallal and P. Sorenson, "Reusing Class-Based Test Cases for Testing Object-Oriented Framework Interface Classes," Journal of Software Maintenance and Evolution: Research and Practice, 2005, Vol. 17, No. 3, pp.169-196.
[20] J. Al Dallal and P. Sorenson, "Generating State Based Testing Models for Object-Oriented Framework Interface Classes," Transactions on Engineering, Computing and Technology, 2006, Vol. 16, pp. 96-102.
[21] 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, 2006, Vol. 16, pp. 90-95.
[22] J. Al Dallal and P. Sorenson, "The Coverage of the Object-Oriented Framework Application Class-Based Test Cases," Transactions on Engineering, Computing and Technology, 2006, Vol. 16, pp. 103-107.
[23] J. Al Dallal, "Testing Object-Oriented Hook Methods," Kuwait Journal of Science and Engineering, 2008, Vol. 35, No. 2.
[24] J. Al Dallal and P. Sorenson, "Estimating the Coverage of the Framework Application Reusable Cluster-Based Test Cases," Journal of Information and Software Technology, 2008, Vol. 50, No. 6, pp 595-604.
[25] J. Al Dallal and P. Sorenson, "Generating Class Based Test Cases for Interface Classes of Object-Oriented Gray-Box Frameworks," International Journal of Computer Science and Engineering, 2008, Vol. 2, No. 3, pp. 135-143.
[26] J. Al Dallal, "Testing Object-Oriented Framework Applications Using $FIST_2$ Tool: a Case Study," International Journal of Computer Systems Science and Engineering, 2008, Vol. 4, No. 2, pp. 119-126.
[27] J. Al Dallal, "Adequacy of Object-Oriented Framework System-Based Testing Techniques," International Journal of Computer Science, 2008, Vol. 3, No. 1, 36-43.
[28] A. Tevanlinna, "Product Family Testing with RITA," Proceedings of the Eleventh Nordic Workshop on Programming and Software Development Tools and Techniques, Turku, Finland, 2004.
[29] M. B. Cohen, M. B. Dwyer, and J. Shi, "Coverage and Adequacy in Software Product Line Testing," Proceedings of the International Symposium on Software Testing and Analysis, 2006 workshop on Role of software architecture for testing and analysis, Portland, Maine, USA, 2006.
[30] M. Grindal, J. Offutt, and S.F. Andler, "Combination Testing Strategies: a Survey," Software Testing, Verification and Reliability, 2005, Vol. 15, No. 3, pp. 167-199.
[31] A. Jaaksi, "Developing Mobile Browsers in a Product Line," IEEE Software, 2002, Vol. 19, No. 4, pp. 73-80.
[32] T. L.Graves, M. J. Harrold, Y. M. Kim, A. Porter, and G. Rothermel, "An Empirical Study of Regression Test Selection Techniques," ACM Transactions on Software Engineering and Methodology, 2001, Vol. 10, No. 2, pp. 184-208.
[33] J. Zheng, "In Regression Testing Selection when Source Code is not Available," Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, Long Beach, CA, USA, 2005.
[34] J. Zheng , B. Robinson , L. Williams , K. Smiley, "Applying Regression Test Selection for COTS-Based Applications," Proceeding of the 28th international conference on Software engineering, Shanghai, China, 2006.
[35] M. Harrold, J. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. Spoon, and A. Gujarathi, "Regression Test Selection for Java Software," Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2001), Tampa, Florida, USA, 2001, pp. 312-326.
[36] G. Rothermel, M. Harrold, and J. Dedhia, "Regression Test Selection for C++ Software," Journal of software testing, Verification, and Reliability, 2000, Vol. 10, No. 6, pp. 77-109.
[37] S. Wilkin and D. Hoffman, "Junit Extensions for Documentation and Inheritance," Proceedings of the 2002 Pacific Northwest Software Quality Conference, Protland, USA, 2002, pp. 71-84.
[38] J. Anvik, S. MacDonald, D. Szafron, J. Schaeffer, S. Bromling and K. Tan, "Generating Parallel Programs from the Wavefront Design Pattern," Proceedings of the 7th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'02), Fort Lauderdale, Florida, 2002.
[39] K. Tan, D. Szafron, J. Schaeffer, J. Anvik, and S. MacDonald, "Using Generative Design Patterns to Generate Parallel Code for a Distributed Memory Environment," Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, USA, 2003, pp. 203-215.
[40] Y. Cheon and G. Leavens, "A Simple and Practical Approach to Unit Testing: the JML and Junit Way," Proc. of the 16th European Conference on Object-Oriented Programming (ECOOP2002), 2002, pp. 231-254.
[41] Jcontract, http://www.parasoft.com/, ParaSoft Corporation, May 2007.
[42] 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.
[43] J. Offutt and J. Pan, "Automatically Detecting Equivalent Mutants and Infeasible Paths," The Journal Of Software Testing, Verification, and Reliability, 1997, Vol. 7, No. 3, pp 165-192.

© 2008 ACADEMY PUBLISHER