Integrating Hook-Based Object-Oriented Framework Testing Techniques

Jehad Al Dallal

***Abstract*—** Object-oriented frameworks provide reusable design, implementation, and testing for a family of software systems that share common features. They are implemented at the framework domain engineering stage and extended at the application engineering stage to build the particular required applications. Places at which the framework is extended are called hooks. These hooks are useful in testing both the framework and its applications. Several non-integrated hook-based testing techniques are introduced to test the frameworks and their applications at different engineering stages and testing levels. This paper discusses the integration of four framework-based testing techniques such that the testing redundancy is minimized and the testing reusability is maximized. The testing techniques are originally introduced to test the framework and hooks during the domain engineering stage, and to re-test the framework and test the framework interface classes during the application engineering stage. Finally, the paper illustrates the design of the tools that support the automation of the integrated techniques.

Keywords — object-oriented framework, object-oriented testing, test case generation, testing automation.

I. Introduction

REUSABILITY is one of the fundamental goals of software engineering. Object-oriented frameworks achieve this goal by providing reusable design, code, and testing for a family of software systems. A framework contains a collection of reusable concrete and abstract classes, and it reduces the cost of a product line (i.e., family of products that share common features) and increases the maintainability of software products [1]. Developers can reuse and extend the design and implementation of a suitable framework to build their particular applications instead of developing them from scratch. Places at which developers can extend the framework and add their own classes are called hooks [2]. Object-oriented framework engineering is divided into separate domain and application engineering tasks. During domain engineering, the framework classes are produced. During application engineering, the users of the framework complete or extend the framework classes to build their particular applications.

To build an application using a framework, application developers create two types of classes: (1) classes that use the framework classes, and (2) classes that do not. Classes that use the framework classes are called Framework Interface Classes (FICs) [1, 3] because they act as interfaces between the framework classes and the second type of the classes created by application developers. Fig. 1 shows the relationship between the framework classes, the hooks, the FICs, and the other application classes. 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, and therefore, they define the FICs and specify the pre-conditions and post-conditions of the FIC methods. Froehlich [2] provides a special purpose language and grammar in which the hook description can be written. The hook description includes the implementation steps and the specifications (i.e., pre-conditions and post-conditions) of the FIC methods.

Manuscript received August 30, 2008: Revised version received December 1, 2008.

Jehad Al Dallal is with Department of Information Science, Kuwait University, P.O. Box 5969, Safat 13060, Kuwait (e-mail: jehad@cfw.kuniv.edu).

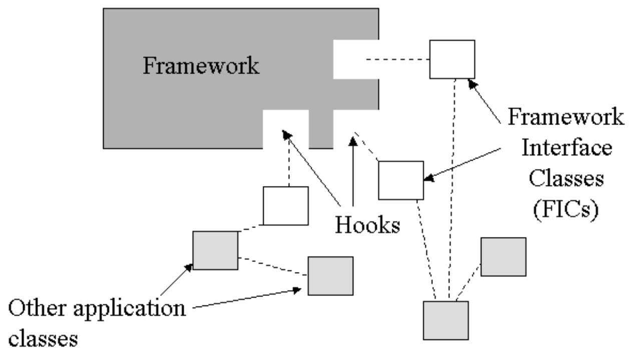


Fig. 1: Framework application classes

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.

Testing has recently been addressed to complete the framework development life-cycle. Testing the framework before instantiating it is essential; otherwise, if the framework contains defects, these defects will be passed on to the applications developed from the framework. Framework defects are hard to discover at the time the framework is instantiated. Therefore, it is important to remove all defects before instantiating the framework. In addition, it is important to verify that the framework hooks are specified correctly. Otherwise, the generated implementations of the hook methods will not function properly. Several techniques are proposed to test the framework reusable code and design (e.g., [4-8]) and to test the framework hooks (e.g., [9, 10]).

Testing the framework increases confidence that the framework is designed and implemented correctly. However, due to the infiniteness of the possible test data, the framework testing does not guarantee the correctness of the framework design and implementation; and, therefore, it is important to reconsider framework testing during the application engineering stage [11]. In addition, testing the framework applications includes testing the implemented FICs, the other classes created by the application developers, and the relations among the application classes. Several techniques are proposed to test the framework applications (e.g., [12-17]).

In a product line, increasing reusability and decreasing redundancy are essential goals. However, researchers have dealt with each of the above framework testing areas in isolation from the others, despite the fact that these testing areas are close to each other. Ignoring the application of testing performed at one stage when testing a related area in another stage in a product line increases the chance of work-redundancy.

In this paper, we propose an integrated environment that considers the testing of the framework and its applications at both domain and application engineering stages. At the domain engineering stage, the integrated environment considers the framework testing at system level and the hook testing. At the application engineering stage, the integrated environment considers the framework re-testing and the FICs testing. The main goal of this integrated environment is to reduce redundancy in framework and application testing and to increase the reusability of the various stages and levels of framework and application testing. The proposed integrated testing environment relies on the testing techniques previously introduced by the author, including [1, 6, 9, and 11].

The paper is organized as follows. Sections II and III overview related work and already existing models used for testing object-oriented frameworks and their applications, respectively. Sections IV, V, VI, and VII introduce the integrated framework-based testing environment and discuss the automation issues related to the four testing techniques integrated in the environment. Finally, Section VIII concludes the paper.

II. RELATED WORK

The environment proposed in this paper integrates and modifies the testing techniques introduced in [1, 6, 9, and 11]. All these technuqies are hook-based ones. This section summarizes the hook notation and already existing techniques for testing object-oriented frameworks and their applications. In addition, this section gives an overview of the other related work in the same testing area.

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. 2 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

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:

specify the bank branches in the system. Finally, the user must specify the *maxPeriod* variable value.

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

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.

B. Testing Frameworks Through Hooks (TFTH)

Al Dallal and Sorenson [6] propose a technique called Testing Frameworks Through Hooks (TFTH) to generate a test suite to test hook-documented object-oriented frameworks. The hook-documented frameworks are those provided with hook descriptions. Hook descriptions give specifications for the FICs and guidelines to implement them. In TFTH testing technique, the test suite is designed to test framework implementation at the system level as well as the framework FICs. The technique uses an extended state model for the FICs and a construction flow graph to model the construction sequence of the hook methods. Round-trip path trees [4] are generated from the FICs state models. The trees and the construction flow graphs are traversed to produce the required test suite.

C. The hook method testing technique

Al Dallal [9] proposes a technique and a supporting tool to build a test suite for the FICs methods. These methods are called hook methods because their implementation and construction process is specified in the hook descriptions. The technique produces different demo implementations for the hook methods using the same construction flow graph used in the TFTH technique. In addition, the technique generates test data for all variables used in the hook method. The test cases are generated using a combination of demo implementations and the test data. Finally, the technique uses the specifications of the hook methods given in the hook descriptions to evaluate the test cases.

D. The framework part test-case-reusing technique

Al Dallal and Sorenson [11] propose 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 testing models proposed in the TFTH technique. The test-case-reusing technique first identifies the non-tested 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 non-implemented 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.

E. Testing framework FICs

In [1, 3], a technique is introduced to generate reusable test cases for the FICs during the domain engineering stage and to apply them to testing the FICs at class-level during the application engineering stage. A technique is introduced to automate the construction of the class-based testing model, using the method specifications provided in the hooks [13]. In addition, a technique called all paths-state is introduced; it uses the constructed testing model to generate the class-based reusable test cases at the domain engineering stage [3]. At the application engineering stage, the application developers may need the flexibility to ignore or modify part of the specifications used to generate the reusable class-based test cases and to add new specifications not covered by the reusable test cases. The technique introduced in [1] shows how to deal effectively with such modifications so that testing becomes easy and straightforward during the application development process.

F. Other related work

Several recent research studies address the problem of object-oriented testing at different levels in general (e.g., [4], [18-21] and [28]). Some testing techniques are specifically proposed to test object-oriented frameworks and their instantiations (e.g., [1, 3-17]).

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. [5] 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 to make sure 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. [7] 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 ensures the inherited framework features work correctly in the context of the instantiation classes that inherit them. Kauppinen et al. [10] propose a criterion to evaluate the hook coverage of a test suite used to test hook methods. RITA [22] is a software tool that supports framework testing and automates the calculation of the hook method coverage measure. Al Dallal and Sorenson [15] propose a methodology to estimate the coverage of the cluster-based reusable test cases for framework instantiations.

The work on testing the software product line and product family is relevant to the problem of testing frameworks. A software product family is a set of software products that share common features [23, 24, 27]. The natural core of a product family is a set of software assets that is reused across products [25]. Variation points are points at which the products of a software family differ (i.e., each product has a different implementation, which is called a variant, for an abstract class associated with a variant point) [26]. In framework-based software product families, the variation points are the hook points, and implementations of the FICs are the variants. Cohen et al. [26] suggest using combination testing strategies (e.g., [29]) to build test cases to test product line variants. Tevanlinna et al. [25] identify and compare four different strategies for modeling product family testing.

III. Testing Models

In this paper, we consider the integration of four framework-related testing areas: testing the framework at system level, testing the hook methods, re-testing the re-used part of the framework at system level, and testing the implemented FICs using reusable test cases. The former two areas are considered during the domain engineering stage, and the other two areas are considered during the application engineering stage. Multiple testing modules are used in [3, 6, 9, 11] to achieve the coverage of the four framework-related testing areas as follows.

A. State Transition Diagram (STD)

A class behavior can be graphically represented in a state transition diagram. 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. An STD consists of nodes and direct links. Each node represents a state and each link represents a transition. Fig. 3 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, and Frozen states to model the states of the object.

In [13], the state transition diagrams of the FICs are constructed automatically using the specifications given in the hook descriptions provided with the framework. The diagram is traversed using an all paths-state coverage technique [1], illustrated below, to determine the sequence of message executions required to build the test cases. These test cases are built once during the framework domain engineering stage and reused each time an application is developed during the application engineering stage to test the implemented FICs.

B. All paths-state tree

At the application engineering 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 already existing state-based specification coverage criteria. In [12], 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.

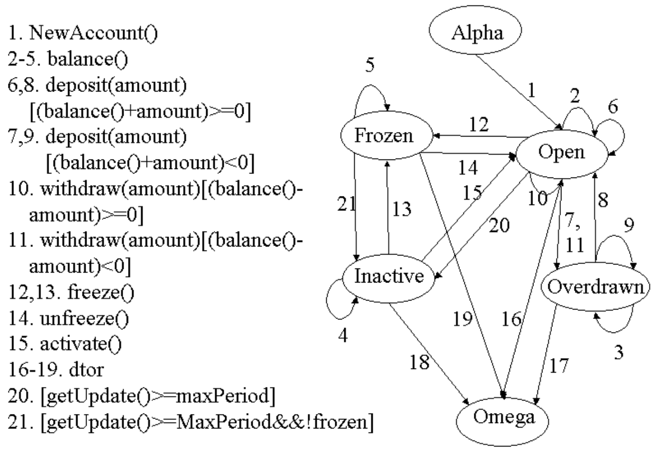

1. NewAccount()
2-5. balance()
6,8. deposit(amount) [(balance()+amount)>=0]
7,9. deposit(amount) [(balance()+amount)<0]
10. withdraw(amount)[(balance()-amount)>=0]
11. withdraw(amount)[(balance()-amount)<0]
12,13. freeze()
14. unfreeze()
15. activate()
16-19. dtor
20. [getUpdate()>=maxPeriod]
21. [getUpdate()>=MaxPeriod&&!frozen]
Alpha
Frozen
Open
Inactive
Overdrawn
Omega
1
2
3
4
5
6
7, 11
8
9
10
12
13
14
15
16
17
18
19
20
21

Fig. 3 The STD of the *NewAccount* object defined in the banking framework hooks.

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. 3. 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.

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.

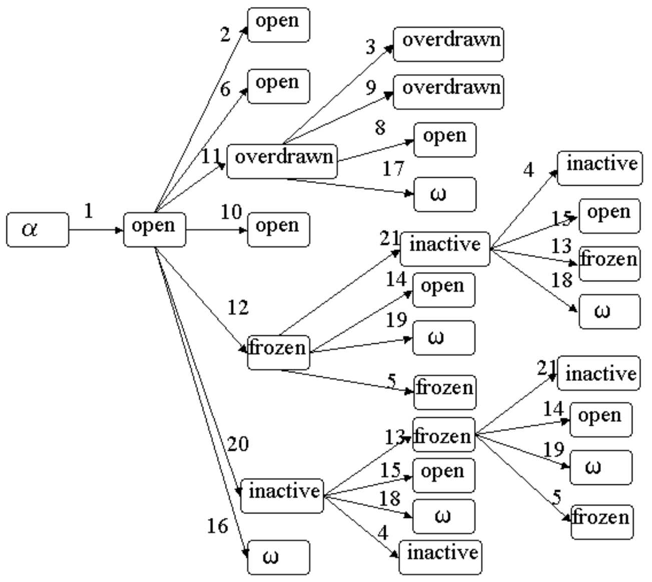


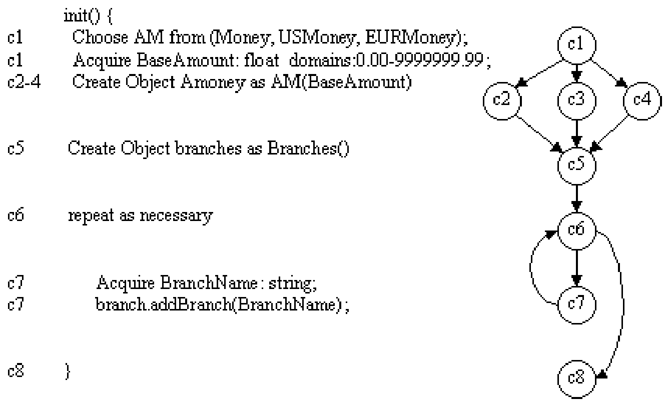
Fig. 4. All paths-state tree of the STD example shown in Fig. 3.

C. Hook State Transition Diagram (HSTD)

An HSTD is a state transition diagram that has two types of links: solid and dotted, which represent transitions associated with explicit and implicit events, respectively. Implicit events are implicit calls for methods (i.e., those caused by calling other methods). The implicit events are modeled in the HSTD such that the different implementations of hook methods that can only be called implicitly are considered when building the test cases. The HSTD is semi-automated using the framework hooks and it is traversed using a round-trip path coverage technique [4] to determine the sequence of message executions required to build the test cases. These test cases are used to test the framework at system-level during the framework domain engineering stage [6]. In addition, these test cases are re-used to test the re-used part of the framework during the application engineering stage [11].

D. Construction Flow Graph (CFG)

The CFG is a graphical representation of the control structure of the construction sequence of the hook method contents. It consists of nodes and direct links. A node in the CFG can be a process, a decision, or a junction node. The process node presents a sequence of hook statements that are uninterrupted by a construction decision or a construction junction. The decision node is a hook method description point where the construction flow diverges. Finally, the hook method description point where the construction flow merges is called the junction node. Fig. 5 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*.


```
        init() {
c1        Choose AM from (Money, USMoney, EURMoney);
c1        Acquire BaseAmount: float  domains:0.00-9999999.99 ;
c2-4      Create Object Amoney as AM(BaseAmount)


c5        Create Object branches as Branches()


c6        repeat as necessary


c7           Acquire BranchName : string;
c7           branch.addBranch(BranchName) ;


c8      }
```

Fig. 5. The CFG of the *init()* method defined in the *Initialize Account* hook

In [6] and [9] the CFG is used to build different implementations of hook methods. In [6], the combinations of these implementations are exercised by the test cases determined using the HSTD, whereas in [9], each implementation of a hook method is exercised to satisfy some well-known method testing coverage criteria, such as domain boundary and equivalence partitioning [4]. In [11], the CFG is used to identify the hook methods that have to be reconsidered when retesting the framework during the application engineering stage.

IV. Testing Framework

In [6], the framework testing starts with building the HSTD for the FICs. The sequences of method executions considered for building the test cases are determined by applying the round-trip path coverage technique. When generating the test cases for the FICs, the testing models for the FICs are covered using all paths-state coverage which is proved to subsume the round-trip path coverage [3]. As a result, to produce test cases that satisfy the coverage required for both testing the framework and testing the FICs, the testing models for the FICs must be covered using the all paths-state coverage technique. The inputs and outputs of the framework testing tool and the testing process are described as follows.

A. Tool Inputs

The modified framework testing tool requires several inputs as follows:

1. *Framework hook descriptions*. The hook *changes* section describes the changes that can be made to develop a hook method. The hook changes section describes the creation of hook methods and their contents (i.e., code statements). In addition, the syntax of the hook changes section includes calling up other hooks and creating classes (i.e., FICs), objects, and properties. The syntax also allows the user to prompt data or select options, and it allows iterating through a set of change statements. For example, the changes section of the Initialize Account hook (see Fig. 2) describes how the constructor method is built: (1) an constructor method is created, (2) a user is asked to choose one of the money classes

defined in the framework, (3) an object of the selected money class is created inside constructor's method block, (4) an object of the framework class Branches is created inside constructor's method block, and (5) the user is prompted for the number of iterations of the repeat loop. In the repeat loop, the user is prompted for a name to be assigned to the *BranchName* variable, and the *addBranch* method is invoked inside constructor's method block. Finally, the user is prompted for a value to be assigned to the *maxPeriod* variable and the *setMaxPeriod* method is called inside constructor's method block.

2. *Illegal behaviors*. Given the hook descriptions, the HSTDs that model the legal behaviors of the FICs can be extracted automatically. The framework tester has to use other framework specification documents or communicate with framework developers to determine the illegal behaviors of the FICs to complete the HSTDs.

B. Tool Outputs

The modified framework testing tool produces several outputs as follows:

1. *Implemented hook methods*. The modified framework testing tool uses the Hook Master tool [2] to produce multiple Java implementations of the hook methods and comments on them with the corresponding pre-conditions and post-conditions specified in the hook description. These implementations of the hook methods are stored in the framework database to be used in the hook testing process.

2. *Framework test cases*. The framework test cases are formed by combining the implementation of the hook methods and the all paths-state test drivers. The test cases are stored in the framework database to be used in the FICs testing and framework re-testing processes.

3. *Test case execution results.* The framework testing tool executes the test cases and uses the Jcontract tool [29] to evaluate the testing results.

C. Testing Process

The modified framework testing process is shown in Fig. 6. In this process, the tester selects the framework to be tested using a browser. The framework is stored in a database that contains the framework code and descriptions of the hooks. In the tool, the hook descriptions are passed to the *Hook Master* tool. The Hook Master tool parsers the hook description and stores it in hook statement objects. The hook statement objects are stored in the framework database and passed as parameters to the *HSTD Master* tool, which analyzes the hook statements, builds the HSTDs, and stores them in a tabular form. A framework tester can edit the HSTD tables to describe the behavior of the FIC in response to illegal events. The HSTD Master tool uses the updated tables to generate the all paths-state Java-coded test cases to be used in constructing the framework test cases.

Simultaneously, the Hook Master tool produces different Java implementations of the hook methods and comments on them with the corresponding pre-conditions and post-conditions specified in the hook description. The pre-condition and post-conditions are written in DbC language [30]. *TCs builder* obtains the all paths-state test cases generated by the HSTD Master tool and the different implementations of the DbC commented hook methods generated by the Hook Master tool, combines them, and produces the framework test cases. The test cases are then stored in the framework database.

The *Test cases executer* module compiles the test cases using the dbc_javac compiler of the *Jcontract* tool [29]. The *Jcontract* compiler checks the DbC specifications in the Javadoc comments, generates instrumented .java files with extra code to check the contracts (i.e., pre-conditions and post-conditions) 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 framework testing tool executes the test cases and uses *Jcontract* tool to automatically check the contracts at runtime, report any violations, and stack trace information in the *Jcontract* GUI Monitor, the *Jcontract* TEXT Monitor, or a file. This helps users determine exactly when and where a violation occurs.

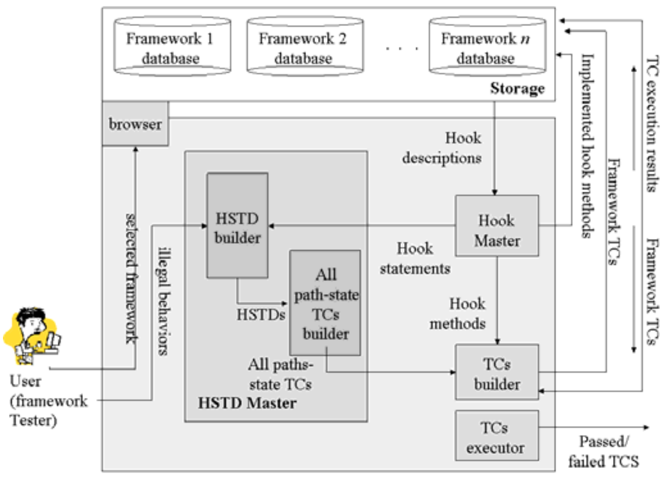


Fig. 6. Modified framework testing process

V. Testing Framework Hooks

In [9], the hook testing process requires building multiple implementations for the hook methods using the CFGs. This step is already performed when testing the framework, and its results are stored in the framework database as illustrated above. These implementations are exercised with the test data generated for the parameters of the hook methods to complete the hook testing process. The inputs and outputs of the framework hooks testing tool and the testing process are described as follows.

A. Tool Inputs

The modified framework hook testing tool requires several inputs as follows:

1. *Selected hook descriptions*. The user of the tool selects the hooks to be tested.

2. *Selected test data generation technique*. The user can select one of the three combination-of-test-data generation techniques implemented in the tool. The tool implements two

test data generator techniques for the variables specified in the hook description, the domain boundary analysis and equivalence partitioning techniques. Combinations of test data are generated using two approaches, boundary-typical and all-combinations. In the boundary-typical approach, the variable under consideration is assigned one of the test data and all other variables are assigned typical values. In the all-combinations approach which is more sophisticated, all combinations of test data are used to generate the required combinations.

3. *Implemented hook methods*. These implemented hook methods are stored in the framework database as a result of the framework testing process described in Section IV.

4. *Framework Code*. The framework code is stored in the database and used together with the test cases to test the hooks and obtain the testing results.

B. Tool Outputs

The modified framework hook testing tool produces several outputs as follows:

1. *Hook methods test drivers*. Each test driver checks the hook method's pre-conditions before enacting the hook method, enacts an implementation of the hook method and checks the post-conditions after enacting the hook method. As a result, a test driver is a class that includes an implementation for the hook method and a *TestHook* method that invokes the hook method and compares the actual results with the expected ones.

2. *Drivers for the test drivers*. The tool generates a driver that enacts the hook methods test drivers.

3. *Testing results.* The framework hook testing tool uses the Junit tool [31] to obtain the testing results.

C. Testing Process

The modified framework hook testing process is shown in Fig. 7. In this process, the user of the tool selects a framework through a browser. The tool loads the names of the available hook descriptions and shows them on the tool's GUI. When the user selects one of these names, the *Hook Method Parser* block of the tool parses the corresponding implemented hook methods produced in the framework testing process described in Section IV. The user can select one of the three combination-of-test-data generation techniques implemented in the tool. When the user selects the combination-of-test-data generation technique, the *Test Cases Builder* block of the tool uses the parsed hook methods and the selected combination of test data generation technique to generate the hook method test cases. Each test case describes an implementation for the hook method. The tool stores the test cases in the framework database. The *Test Drivers Builder* block of the tool enacts the test cases and generates corresponding test drivers. Each test driver is a Java implementation of a test case. The *Test Drivers Builder* block of the tool also generates a driver for the test drivers. Finally, the *Test Drivers Executor* block of the tool invokes the Junit tool [31] that invokes the *TestHook* method of each test driver and reports the testing results. The modified hook method testing tool tests the hook methods involved in the selected hook description. To test all the hook methods, the user of the tool has to select all the hook descriptions one at a time. When a hook description is selected, only the involved FICs are generated. The framework with the generated FICs are considered a framework application. Typically, it is not required to implement all hook methods to build a framework application.

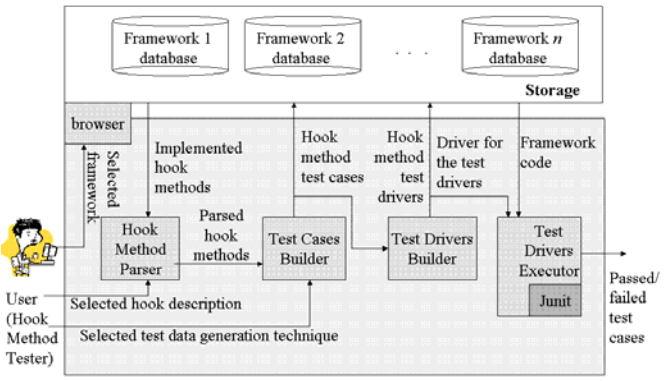


Fig. 7. Modified framework hooks testing process

VI. RE-TESTING THE FRAMEWORK USED PART

In [11], the re-testing framework used part process assumes that the round-trip path coverage is applied in the TFTH to produce the test cases. In the modified environment, as discussed above, the all paths-state coverage is applied to produce the test cases. In [11], when the application developer decides not to use a transition modeled in the HSTD, the round-trip path tree has to be remodeled such that all reachable transitions remain connected in the tree. This step becomes unnecessary when using the all paths-state tree because the later one is constructed in such a way that if a transition is deleted, the remaining transitions remain reachable in the tree. Therefore, in our modified environment, reusing the test cases generated by using the all paths-state coverage approach will not cause incompatibility problems; instead it eases the required testing process. The inputs and outputs of the framework re-testing tool and the testing process are described as follows.

A. Tool Inputs

The modified framework re-testing tool requires several inputs as follows:

1. *Implemented hook methods*. These implemented hook methods are stored in the framework database as a result of the framework testing process described in Section IV.

2. *Framework test cases*. These implemented hook methods are also stored in the framework database as a result of the framework testing process described in Section IV.

3. *Implemented FICs*. These classes are part of the framework application developed by a framework user.

4. *Framework code*. The framework code is stored in the database and used together with the test cases to re-test the used part of the framework and obtain the testing results.

B. Tool Outputs

The modified framework re-testing tool produces several outputs as follows:

1. *Applicable test cases*. These test cases are subset of the test cases generated to test the framework at the framework engineering stage and they have to be applied to retest the framework during the application engineering stage.
2. *Testing results.* The framework re- testing tool uses the Jcontract tool [29] to obtain the testing results.

C. Testing Process

The modified framework re-testing process is shown in Fig. 8. In this process, 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 the framework testing tool, and the implemented hook method data files produced by the framework testing tool. The *Test Case Parser* module of the framework re-testing 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.

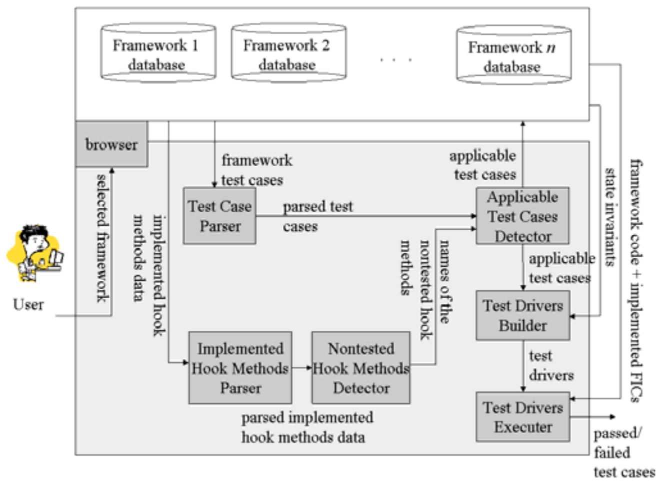


Fig. 8. Modified framework used part re-testing process

The parsed implemented hook method data are used by the *Nontested Hook Methods Detector* module of the tool to identify the nontested hook methods. 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. 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 [30] 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. The Jcontract compiler checks the DbC specifications in the Javadoc comments, generates instrumented .java files with extra code to check the contracts 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.

VII. TESTING FRAMEWORK INTERFACE CLASSES

In [6], the cases generated to test the framework during the domain engineering stage were built using the round-trip path coverage approach. Since this coverage is not suitable for testing FICs, in [3], special reusable class-based test cases are built during the domain engineering stage and applied during the application engineering stage to test the implemented FICs. These test cases are generated using the same testing models used for the framework test cases. However, the testing models are covered using the all paths-state covering approach. In our modified environment, the all paths-state coverage is applied to generate the test cases to test the framework. Therefore, the same test cases can be used also to test the FICs. The only difference would be in the ways in which these test cases are applied. Ref [1] discusses how these test cases can be applied effectively. The inputs and outputs of the tool that generates the test cases for the FICs and the test case generation process are described as follows.

A. Tool Inputs

The tool requires several inputs at the framework development stage as follows:

1. *Framework hooks*. Framework hooks define the specifications (i.e., preconditions and postconditions) of the FIC methods introduced by the hooks. These method specifications are used to synthesize the state-based testing model of the FIC at the framework development stage. In addition, they are used as test oracles at the application development stage.
2. *Non-event-driven transitions*. Non-event-driven transitions cannot be synthesized automatically using the algorithms illustrated in [13]. The user of the tool has to determine the source and destination states of the non-event-driven transitions. The tool automatically produces the predicates of the transitions.
3. *Predicate implementation*. Transitions of the FIC synthesized testing model can be associated with predicates that have to be satisfied to execute the transitions. The predicates can be as simple as a variable definition or they can involve defining a large data structure for which it is difficult to generate code to satisfy the predicate. The user of the tool has to provide the code required to satisfy the predicates of the transitions at the framework development stage. Writing the pieces of code that implement complex predicates can be a

costly task; however, this cost cannot be avoided in any state-based testing technique. The good news is that the implementation of the predicates is provided just once at the framework development stage and reused each time an application is developed at the application development stage. In most situations, the original investment can be recouped after producing a few framework applications.

B. Tool Outputs

At the framework development stage, the tool has several outputs. These outputs are used later at the application development stage to test the framework applications. The outputs are as follows.

1. *Class state-based testing model*. The tool synthesizes the class state-based testing models of the FICs at the framework development stage.
2. *Model checking report*. The tool checks that the class state-based testing model has one entry and one exit state and each state can be reached from the entry state. It then reports the checking results.
3. *FIC test drivers*. The tool uses the class state-based testing models of the FICs to generate test drivers using the all paths-state coverage technique [1]. The test drivers are executed later at the application development stage to test the implemented FICs in the framework applications.
4. *Stubs*. The tool analyzes the hook descriptions and uses the information provided in the changes section of the hook description to determine and generate the stubs. These stubs are required at the application testing stage to isolate the FICs. The developed prototype version of the tool does not produce the stubs; instead, the user of the tool has to provide the stubs.

C. Testing Process

Fig. 9 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 preconditions and postconditions 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 a text and stores the text in a file in the framework database. The user can use the *Model Checker module* of the tool to check the correctness of the model.

The *All paths-state test drivers builder* component of the 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, it 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.

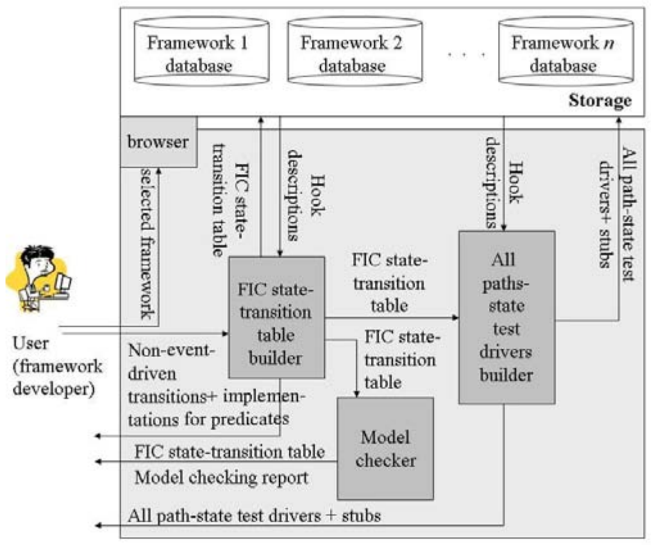

Framework 1 database
Framework 2 database
. . .
Framework n database
Storage
browser
selected framework
FIC state-transition table
Hook descriptions
Hook descriptions
All path-state test drivers+ stubs
User (framework developer)
FIC state-transition table builder
FIC state-transition table
All paths-state test drivers builder
FIC state-transition table
Non-event-driven transitions+ implementations for predicates
Model checker
FIC state-transition table
Model checking report
All path-state test drivers + stubs

Fig. 9. The FIC test case generation process

VIII. Conclusions

This paper introduces an integrated environment for testing object-oriented frameworks and their applications. The environment integrates four testing processes in such a way that redundant testing efforts are reduced. The main reductions are summarized as follows: (1) the same testing models (i.e., HSTD and CFG) are used in all the processes; (2) the same implementations of the hook methods are used in the framework and hook testing processes; (3) the same test cases are used differently in the framework testing, framework re-testing, and FICs testing processes; and (4) the applicable test cases are not required to be modified in the framework re-testing process. On the other hand, the number of test cases to be generated and managed in the framework testing process is enlarged because we propose using all paths-state coverage, which subsumes the round-trip path coverage applied originally. However, this modification allows using the same test cases in two other testing processes.

Acknowledgment

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

References

[1] 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 Practise*, 17(3), 2005, pp. 169-196.
[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] J. Al Dallal, Class-Based Testing of Object-Oriented Framework Interface Classes, *Ph.D. Thesis, University of Alberta, Department of Computing Science*, 2003.
[4] R. Binder, *Testing object-oriented systems*, Addison Wesley, 1999.
[5] 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.

[6] 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, 2002, pp. 231-236.

[7] 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)*, 32(1es), 2000, pp. 7-12.

[8] J. Al Dallal, Adequacy of object-oriented framework system-based testing techniques, *International Journal of Computer Science,* 3(1), 2008, pp. 36-43.

[9] J. Al Dallal, Testing object-oriented hook methods, *Kuwait Journal of Science and Engineering*, 35(1B), 2008, pp. 103-122.

[10] 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.

[11] J. Al Dallal and P. Sorenson, Testing software assets of framework-based product families during application engineering stage, *Journal of Software*, 3(5), 2008, pp. 11-25.

[12] 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*, 16, 2006, pp. 90-95.

[13] J. Al Dallal and P. Sorenson, Generating state based testing models for of object-oriented framework interface classes, *Transactions on Engineering, Computing and Technology*, 16, 2006, pp. 96-102.

[14] J. Al Dallal and P. Sorenson, The coverage of the object-oriented framework application class-based test cases, *Transactions on Engineering, Computing and Technology*, 16, 2006, pp. 103-107.

[15] J. Al Dallal and P. Sorenson, Estimating the coverage of the framework application reusable cluster-based test cases, *Journal of Information and Software Technology*, 50(6), 2008, pp 595-604.

[16] J. Al Dallal, Testing object-oriented framework applications using FIST$_2$ tool: a case study, *International Journal of Computer Systems Science and Engineering*, 4(2), 2008, pp. 119-126.

[17] J. Al Dallal and Paul Sorenson, Generating class based test cases for interface classes of object-oriented gray-box frameworks, *International Journal of Computer Science and Engineering,* 2(3), 2008, pp. 135-143.

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

[19] L. C. Briand, Y. Labiche, and M. Sówka, Automated, contract-based user testing of commercial-off-the-shelf Components, *Pro-ceedings of the 28th International Conference on Software Engineering (ICSE)*, Shanghai, China, 2006.

[20] L. Gallagher and J. Offutt, Automatically testing interacting software components, *Workshop on Automation of Software Test (AST 2006)*, Shanghai, China, 2006.

[21] L. Gallagher, J. Offutt, and A. Cincotta, Integration testing of object-oriented components using finite state machines, *Journal of Software Testing, Verification and Reliability*, 17(1), 2007, pp. 215-266.

[22] A. Tevanlinna, Product family testing with RITA, *Proceedings of the Eleventh Nordic Workshop on Programming and Software Development Tools and Techniques*, Turku, Finland, 2004.

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

[24] M. Jaring, Software Product Family Architectures - Engineering Run-Time Variability Dependencies in an FPGA-based Signal Processing Board, *WSEAS Transactions on Information Science and Applications*, 1(1), 2004, pp. 240-245.

[25] A. Tevanlinna, J. Taina, and R. Kauppinen, Product family testing: a survey, *ACM SIGSOFT Software Engineering Notes,* 29(2), 2004, pp. 12-18.

[26] 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.

[27] F. Ahmed, L. Capretz and M. A. M. Capretz, Framework for Version Control & Dependency Link of Components & Products in a Software Product Line, *WSEAS Transactions on Computers*, Vol. 3, No. 6, pp. 1782-1787, Dec. 2004.

[28] Sani, N. F., Zin, A. M., Idris, S., and Shukur, Z., Designing an understanding and debugging tool (UDT) for object-oriented programming language. *In Proceedings of the 4th WSEAS international Conference on Artificial intelligence, Knowledge Engineering Data Bases*, Salzburg, Austria, February 13 - 15, 2005.

[29] Jcontract, November 2008, http://www.parasoft.com/jsp/products/home.jsp?product=Jcontract, ParaSoft Corpo-ration.

[30] B. Meyer, Design by contracts, *IEEE Computer*, 1992, Vol. 25(10), 40-52.

[31] Junit 2008. http://junit.sourceforge.net/.