Integrated Object-Oriented Framework-Based Testing Environment

JEHAD AL DALLAL
Department of Information Science
Kuwait University
P.O. Box 5969, Safat 13060
KUWAIT

Abstract: - Object-oriented frameworks provide reusable design, implementation, and testing for a family of software systems. Several non-integrated testing techniques are introduced to test the frameworks and their applications at different engineering stages and testing levels. This paper introduces an environment in which four framework-based testing techniques are integrated. 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. The integration is proposed in such a way that the testing redundancy is minimized and the testing reusability is maximized.

Key-Words: - object-oriented framework, object-oriented testing, test case generation.

1 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]. The classes that directly use or inherit the framework classes are called Framework Interface Classes (FICs) [1, 3]. 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.

Testing, a time consuming and labor intensive development task, 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 2 and 3 overview related work and already existing models used for testing object-oriented frameworks and their applications, respectively. Section 4 introduces the integrated framework-based testing environment. Finally, Section 5 concludes the paper.

2 Related Work

The environment proposed in this paper integrates and modifies the testing techniques introduced in [1, 6, 9, and 11]. This section summarizes the 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.

2.1 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.

2.2 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.

2.3 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.

2.4 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.

2.5 Other related work

Several recent research studies address the problem of object-oriented testing at different levels in general (e.g., [4] and [18-21]). 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]. The natural core of a product family is a set of software assets that is reused across products [24]. 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) [25]. In framework-based software product families, the variation points are the hook points, and implementations of the FICs are the variants. Cohen et al. [25] suggest using combination testing strategies (e.g., [26]) to build test cases to test product line variants. Tevanlinna et al. [24] identify and compare four different strategies for modeling product family testing.

3 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.

3.1 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. 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] 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.

3.2 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].
3.3 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. 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.

4 Integrated Framework-Based Testing Process

This section provides an environment in which the processes of testing the framework and the hooks during the domain engineering stage and re-testing the used part of the framework and testing the FICs during the application engineering stage are integrated. The modified processes that achieve the effective integration in terms of reducing testing redundancy and increasing testing reusability are illustrated as follows.

4.1 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 rest of the testing process is the same as in the TFTH technique [6]. In this process, the CFGs of the hook methods are constructed and used to build multiple implementations for the hook methods. The combinations of the implementations are exercised by the test cases.

The modified framework testing process is shown in Figure 1. In this process, the test cases are stored in the framework database to be used in the FICs testing and framework re-testing processes. In addition, the multiple implementations of the hook methods are stored in the framework database to be used in the hook testing process.

4.2 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 as shown in Figure 2.

4.3 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 modified framework re-testing process is shown in Figure 3. In this process, the test cases generated during the domain engineering stage and stored in the framework database are parsed, and the applicable ones are determined and executed.

![Figure 3. Modified framework used part re-testing process](image)

4.4 Testing FICs

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.

5 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 W101/06.

References


