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

Adequacy of Object-Oriented Framework System-Based Testing Techniques

Jehad Al Dallal

***Abstract*—**An application framework provides a reusable design and implementation for a family of software systems. If the framework contains defects, the 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 this paper, two measures for the adequacy of an object-oriented system-based testing technique are introduced. The measures assess the *usefulness* and *uniqueness* of the testing technique. The two measures are applied to experimentally compare the adequacy of two testing techniques introduced to test object-oriented frameworks at the system level. The two considered testing techniques are the New Framework Test Approach and Testing Frameworks Through Hooks (TFTH). The techniques are also compared analytically in terms of their coverage power of object-oriented aspects. The comparison study results show that the TFTH technique is better than the New Framework Test Approach in terms of usefulness degree, uniqueness degree, and coverage power.

***Keywords*—**Object-oriented framework, object-oriented framework testing, test case generation, testing adequacy.

I. Introduction

SOFTWARE testing is the process of executing a program with the intent of finding errors. Testing is a time-consuming and costly ongoing activity during the application software development process. In theory, testing cannot prove the absence of errors, but it increases the level of confidence in the developed 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 preconditions, and the expected output. To test an object-oriented application, four main testing levels have to be exercised, including method testing, class testing, cluster testing, and system testing [1]. At the method testing level, the method responsibilities are considered. At the class testing level, the intraclass interactions and superclass/subclass interactions are examined. At the cluster testing level, the collaborations and interactions between the system classes are exercised. Finally, at the system testing level, the complete integrated system is exercised, usually based on acceptance testing requirements.

Manuscript received May 12, 2007. The author would like to acknowledge the support of this work by Kuwait University Research Grant WI01/06.

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

An object-oriented framework is the reusable design and implementation of a system or subsystem [2]. It contains a collection of reusable concrete and abstract classes. The framework design provides the context in which the classes are used. The framework itself is not complete. Users of the framework are expected to complete or extend the framework to build their particular applications. Places at which users can add their own classes are called hook points [3]. Typically, hooks are associated with problem domain classes. Problem domain classes are representations of external entities or concepts that are necessary for the implementation-independent models of the system. For example, in an order-processing system, Customer, Order, and Product are problem domain classes; LinkedList is not. When the framework is used to build an application, hooks are used to build classes that extend or use the problem domain classes. These classes are called framework interface classes (FICs). The methods inside FICs are called hook methods. Instances of FICs are called framework interface objects. Fig. 1 shows the relationship between the framework's problem domain classes, the hooks, and the FICs.

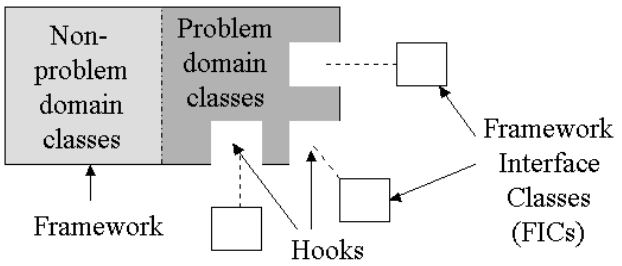

Fig. 1 Framework instantiation classes

Framework defects are passed on to the applications developed from the framework, and they might show up under certain uses. Frameworks can be used in many different ways, and therefore, they require the application of a lot of test cases. Software testing is a time consuming and labor-intensive task. In order to reduce the software testing cost while maintaining the same or better software quality, framework developers aim at applying testing techniques that are adequate for object-oriented frameworks. Test techniques adequate for testing object-oriented frameworks at system level produce test cases that are unique (i.e., not redundant), effective in terms of error detection, and focused in terms of testing behaviors anticipated to be exercised by framework users. A redundant test case is a one that its testing coverage is

a subset of the coverage of other prebuilt test cases. Reducing the overlap between the coverage areas of the test cases reduces the number of redundant test cases and consequently reduces the testing time and cost. Similarly, focusing the testing on the behaviors anticipated to be exercised by framework users and ignoring other behaviors added for debugging purposes or to apply object-oriented conventions reduces the testing time and cost. *Useful* test cases are the ones generated to test a framework's anticipated behaviors. As a result, framework developers aim at avoiding the use of testing techniques that generate test cases of low usefulness or high redundancy degrees. Building and applying useless or redundant test cases consume testing time and effort and do not add any testing value.

Several techniques are proposed to test object-oriented frameworks and their applications (e.g., [4,5,6,7,8,9,10,11, 12]). Among the framework testing techniques, the *New Framework Test* [4] and *Testing Frameworks Through Hooks (TFTH)* [7] are the only techniques introduced to test object-oriented frameworks at the system level.

In this paper, two measures are introduced to assess the usefulness and uniqueness of a testing approach. In addition, the two framework system-level testing techniques, New Framework Test and TFTH, are compared analytically and experimentally. The analytical comparison is based on the testing techniques' power in covering the object-oriented aspects and aims at giving insights on the adequacy of the testing techniques in terms of object-oriented-specific error detection power. The experimental comparison aims at evaluating the adequacy of the testing techniques in terms of their usefulness and uniqueness degrees. Two frameworks are considered in the experimental comparison study: the Client-Server Framework (CSF) [3] and the WaveFront Pattern framework [13]. The comparison study results show that the TFTH technique is better than the New Framework Test Approach in terms of object-oriented aspects coverage power, usefulness, and uniqueness.

The paper is organized as follows. Section II discusses the related work. Section III introduces the usefulness and uniqueness metrics. Sections IV and V, respectively, compare the two considered framework testing techniques analytically and experimentally. Finally, Section VI provides conclusions and a discussion of future work.

II. Related Work

Several techniques are proposed to test object-oriented frameworks and their applications. In this paper, two of these testing techniques are considered: the New Framework Test Approach and the TFTH.

A. The New Framework Test Approach

Binder [4] introduces a testing approach called the New Framework Test to develop test cases for a framework that has few, if any, instantiations. In this approach, four likely types of defects are checked: incomplete/missing behavior or representation, broken association constraints, control defects, and infrastructure code defects. The approach suggests building a demonstration application that provides a minimal implementation level for each use case. Test cases have to be developed to test the demo application using three testing techniques: (1) the *Extended Use Case Test*, (2) the *Class Association Test*, and (3) transition coverage for a state machine (*N+ Test* technique) or branch coverage on all sequence diagrams. The Extended Use Cases technique develops test suites to cover application input/output relationships. The Class Association Test technique checks the implementation of class associations. The N+ Test technique uses a state-based model to construct a graph called a round-trip path tree, which shows all round-trip paths. Round-trip paths are transition sequences that start and end with the same state and simple paths from the alpha to the omega states. A simple path includes only an iteration of a loop, if a loop exists in a sequence. Each path, from the root of the tree to a terminal node, presents a possible test case. The N+ coverage also requires building a test case for each illegal or unspecified event of a state. Finally, all guards associated with the transitions have to be exercised at least once in a test case. The N+ strategy covers guard faults, missing transitions, sneak paths (extra transitions), incorrect actions (wrong or missing), incorrect resultant states, missing states, and corrupt resultant states.

B. The TFTH Technique

TFTH is a testing technique that tests frameworks at the 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. Hook descriptions specify the behaviors of the FICs, and they are used to construct the FIC Hook State Transition Diagram (HSTD) automatically. 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. A transition is an allowable two-state sequence caused by an event. An event is a method call. 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

Hook descriptions define how to construct FIC methods. These methods are called hook methods. 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.

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 implementation of one or more FICs and the framework code.

The TFTH technique generates a framework test suite 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], which guarantees covering each transition at least once. The tree shows all the round-trip paths. Round-trip paths are transition sequences that start and end with the same state and simple paths from alpha to omega states. A simple path includes only an iteration of a loop, if a loop exists in a sequence. In this step, the dotted links in the HSTD are represented by dotted links in the round-trip path tree.
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 a sequence.
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

Binder [14] suggests that the testing of framework instantiations should be based on system requirements. The new classes and objects developed by the instantiation developer must be individually tested. Moreover, cluster testing should be applied to verify that the developer objects are making correct use of the framework code. In this step, the framework test suite can be extended to test the instantiation extensions. Tsai et al. [5] discuss the issues of testing instantiations developed with design patterns using object-oriented frameworks. Framework developers should test that the extensible patterns allow the instantiation developer to extend its functionality. The instantiation designers should verify that the extension points are properly coded and tested. The paper introduces a technique to generate scenario templates that can be used to generate different types of cluster-based test scenarios. These test scenarios are used to test sequence constraints on the interactions between framework objects and custom objects. Wang et al. [6] propose providing the framework with reusable test cases that can be applied at the instantiation development stage.

Al Dallal and Sorenson [8,15,16,17] propose a technique to test the FICs at the class level using reusable test cases built during the framework development stage. The testing is performed in four steps. During the framework development stage, in the first step, the specifications of the FIC methods are used to synthesize the FIC class-based testing model. In the second step, the model is used to generate the reusable class-based test cases for the FIC. During the framework instantiation development stage, in the third step, the test cases are used to test the implemented FICs. Finally, in the fourth step, the specifications of the FIC methods are used to evaluate the results of the test cases. Al Dallal [12] proposes a technique to test the frameworks's hook methods. The technique builds demo implementations for the hook methods and test suites to test the demo implementations. Kauppinen et al. [9] propose a criterion to evaluate the hook coverage of a test suite used to test hook methods. The hook method coverage is defined as the structural coverage (e.g., statement coverage) of a hook method implementation provided by the test cases that reach the method. RITA [10] is a software tool that supports framework testing and automates the calculation of the hook method coverage measure. The user of the tool has to provide implementations for the hook methods.

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 shares common features [18]. The natural core of a product family is a set of software assets that is reused across products [19]. 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 class associated with a variant point) [11]. In framework-based software product families, the variation points are the hook points, and implementations of the FICs are the variants. McGregor [20] suggests testing the product line core assets before using them in building the product. Methods of classes associated with variation points can be tested using pre-built variants produced by the product line organization. As does McGregor [20], Cohen et al. [11] suggest using combination testing strategies [21] to build test cases to test product line variants.

III. The Usefulness and Uniqueness Measures

In practice, not all possible framework object behaviors are expected to be exercised in the framework instantiations. Some of the behaviors can be added to support the polymorphic behaviors (i.e., a method in a superclass that is overridden in all of its subclasses). Other behaviors can be added to access instance variables and are neither used in the framework nor expected to be accessed by the application developers. Some sequences of behaviors might be possible, but they are not expected to be exercised in any of the possible framework instantiations. Some of the behaviors can be added for debugging purposes, and thus, are not required to be tested. Finally, some possible class associations can never be expected to occur in any of the framework instantiations.

When testing object-oriented frameworks at system level, framework anticipated behaviors provided through the problem domain classes should be considered. Other nonanticipated behaviors added for debugging purposes or to apply object-oriented conventions can be ignored because they are not expected to be exercised when the framework is used to build applications. *Useful* test cases are the ones generated to test a framework's anticipated behaviors.

Software testing is a time-consuming and labor-intensive task. To reduce the testing time and effort, software developers aim at avoiding the use of testing techniques that generate test cases of low usefulness degree.

Studying the usefulness degree $US_{tc}(A)$ of a testing technique in testing an application A can be conducted experimentally by (1) identifying the set $S_{behaviours}(A)$ of system under test anticipated behaviors, (2) applying the testing technique to build the set of test cases $S_{tc}(A)$ for application A, (3) identifying the set of test cases $S_{contributing\ tc}(A)$ that contribute to testing the anticipated behaviors identified in Step 1, and (4) calculating the ratio US_{tc} of the test cases that contribute to testing the anticipated behaviors to the total number of built test cases. The usefulness degree of a testing technique in testing an application A is defined formally as follows:

$$US_{tc}(A) = \frac{\left|S_{contributing\ tc}(A)\right|}{\left|S_{tc}(A)\right|}$$

The usefulness degree ranges within the interval [0,1] where 0 indicates that none of the test cases is useful for testing the system's anticipated behaviors and 1 indicates that all of the test cases contribute to testing the system's anticipated behaviors. Software testers aim at applying testing techniques that generate test cases of high usefulness degree to test applications at the system level, because these testing techniques are more focused on testing the system's anticipated behaviors.

In some testing approaches, several testing techniques are applied. Each testing technique has a coverage criterion to be satisfied when generating the test cases. The testing criteria of different testing techniques can overlap. Having an overlap, between the testing criteria of different testing techniques, results in producing redundant test cases. Given a certain testing coverage criterion, a redundant (i.e., not unique) test case is one that its testing coverage is a subset of the coverage of other prebuilt test cases. For example, if the coverage criterion requires exercising lines of code 1-10, a test case exercises lines 1-5, and another test case exercises lines 6-7, a test case that exercises lines of code 4-6 is redundant because its testing coverage is a subset of the coverage of the other two test cases. Typically, software developers aim at avoiding the use of a combination of testing techniques that result in producing a high percentage of redundant test cases. Building and applying redundant test cases consume testing time and effort and do not add any testing value.

Studying the uniqueness degree $UN_{tc}(A,S_{techniques})$ of a set $S_{techniques}$ of testing techniques involved in a testing approach used for testing an application A can be conducted experimentally by (1) identifying the coverage criteria imposed by each testing technique and constructing an empty set of test cases $S_{coverage\ criterion}$ for each coverage criterion, (2) applying each testing technique to build the set $S_{tc}(A,S_{techniques})$ of test cases, (3) constructing the set $S_{redundant\ tc}(A,S_{techniques})$ of redundant test cases by applying the Identifying Redundant Test Cases algorithm given in Figure 2, and (4) calculating the ratio of the number of redundant test cases to the total number of built test cases. The uniqueness degree of a testing approach applied for testing an application A is defined formally as follows:

$$UN_{tc}(A, S_{techniques}) = 1 - \frac{\left|S_{redundant\ \ tc}(A, S_{techniques})\right|}{\left|S_{tc}(A, S_{techniques})\right|}$$

for each test case *tc* in $S_{tc}(A,S_{techniques})$ *do*
  for each set $S_{coverage\ criterion}$ *do*
    if the coverage of the test case *tc* is a subset of the coverage of the test cases included in $S_{coverage\ criterion}$ *then*
      $S_{redundant\ tc}(A,S_{techniques}) = S_{redundant\ tc}(A,S_{techniques}) \cup tc$
    else
      $S_{coverage\ criterion} = S_{coverage\ criterion} \cup tc$

Fig. 2 Identifying Redundant Test Cases algorithm

Since $0 \leq S_{redundant\ tc}(A,S_{techniques}) < S_{tc}(A,S_{techniques})$, the value of $UN_{tc}(A,S_{techniques})$ is always greater than 0 and less than or equal to 1. The uniqueness degree 1 indicates that all the test cases are unique. Software testers aim at applying testing approaches that have high uniqueness degree to test applications at the system level, because these testing approaches are more efficient in terms of testing time.

IV. The Analytical Comparison

Table I summarizes the relative coverage power of the New Framework Test approach and the TFTH technique. The two testing techniques cover the framework system use cases using two different techniques. The New Framework Test uses the Extended Use Case Test, a technique used for testing any object-oriented application at the system level. The TFTH technique uses hooks, a specific technology introduced for object-oriented frameworks. In this testing technique, the FICs created at hook points are tested, and they are used to test the framework at the system level. Some hook descriptions specify the extensibility of the framework, and therefore, the TFTH technique uses such hook descriptions to build test cases that can be used at the instantiation development stage to test the framework's extensibility.

The New Framework Test approach is better in terms of class-state transition coverage, because it uses the N+ Test, a specific class-based testing technique. However, some of the state transitions are not anticipated to be exercised by the framework instantiations, as discussed in Section I. Therefore, the TFTH is better in terms of focusing the test on achievable state transitions.

The New Framework Test approach uses the Class Association Test, a specific testing technique for class associations. However, the testing technique is used to test only the multiplicity of the associations between framework and instantiation classes. The TFTH technique tests, in addition, the multiplicity of the associations between the framework and instantiation classes. The New Framework Test approach tests the inheritance relation between the

framework classes. The TFTH technique covers, in addition, the inheritance of the framework classes. Finally, the TFTH technique addresses testing polymorphic behaviors specified in the option hooks. The testing of polymorphic behaviors is not addressed in the New Framework Test approach.

TABLE I
COVERAGE POWER COMPARISON BETWEEN THE NEW FRAMEWORK TEST APPROACH AND THE TFTH TECHNIQUE

Coverage criterion	New Framework Test approach	TFTH technique
System use case coverage	All use cases are covered using Extended Use Case Test.	All use cases are covered using hooks.
Class state transition coverage	All state transitions are covered.	Most of reachable state transitions are covered.
Class association coverage	Multiplicity of associations among framework classes are covered.	1. Multiplicity of associations between instantiation and framework classes are covered. 2. Most of the multiplicity of associations among framework classes are covered.
FIC state transition coverage	Not applicable	All state transitions of the FICs are covered.
Inheritance behavior coverage	All inheritance behaviors among framework components are covered.	1. All expected to be used inheritance behaviors of the framework components are covered. 2. Most of inheritance behaviors among framework components are covered.
Polymorphic behavior coverage	Not covered	Polymorphic behaviors related to options in hook option statements are covered.
Extensibility coverage	Not considered	Test cases are generated to test the extensibility of the framework.

V. THE EXPERIMENTAL COMPARISON STUDY

The New Framework Test approach and the TFTH technique were used to generate test cases for two frameworks: the CSF and the WaveFront Pattern. The test cases are compared in terms of their usefulness and uniqueness.

A. Generating Test Cases for the CSF

The CSF is a communications framework written in Java and developed to support the building of relatively small applications that require client-server or peer-to-peer communication support. The CSF also provides persistent storage capabilities and can handle communications over a TCP/IP connection using a model similar to email. The CSF deals with synchronous and asynchronous messages sent between remote objects. The framework code consists of 38 classes and about 1.4K lines of code (without comments/blank lines).

Both the New Framework Test approach and the TFTH technique are applied to generate test cases for the CSF. As mentioned in Section II, the New Framework Test approach uses three testing techniques. The results of applying the three testing techniques are shown in the second, third, and fourth columns of Table II. The fifth column summarizes the results of applying the three techniques. The last column summarizes the results of applying the TFTH technique. The first row of Table II lists the names of the applied testing techniques. The second row shows the number of test cases generated using each of the testing techniques. The third row reports the number of useful test cases. The fourth and fifth rows report the usefulness and uniqueness degrees of the testing techniques. The sixth row reports the number of classes exercised directly by the test cases. The seventh row reports the number of useful test cases that are covered using the testing technique but are not covered using the other testing technique. Finally, the last row reports the number of test cases generated to test the extensibility of the framework.

The results of applying the New Framework Test approach show that it has a low (0.288) usefulness degree. This is due to that fact that N+ Test is a class-based technique applied for each class in the framework. Some of the framework classes are not directly accessed by the framework users. For such classes, some sequences of behaviors might be possible, but they are not expected to be exercised in any of the possible framework instantiations. As a result, building and applying test cases to test such sequences is ineffective in terms of time

and effort. The same applies for the test cases generated using the *Class Association Test* technique because some possible class associations covered by the testing technique can never be expected to occur in any of the framework instantiations. All the test cases generated using the Extended Use Case Test are useful, because the testing technique focuses only on cases expected to be exercised in the framework instantiation. On the other hand, the results of applying the TFTH technique show that all the generated test cases are useful. This is due to the fact that the TFTH technique performs the testing through the FICs, and therefore, only possible cases to be exercised are considered.

The results given in Table II show experimentally that the coverage areas of the three testing techniques used in the New Framework Test approach overlap (i.e., they are not orthogonal). As a result, some of the testing work performed using the New Framework Test approach is redundant. Generating and applying redundant test cases is ineffective in terms of time and effort. On the other hand, the results of applying the TFTH technique show that none of the generated test cases are redundant. This is due to the fact that the TFTH technique uses one technique to generate the test cases.

The results given in Table II show that only a few of the useful test cases (2.2% of the useful test cases) generated using the New Framework Test approach is not covered using the TFTH technique. Most of these test cases are used to test the associations between framework classes. This indicates that the New Framework Test approach is better than the TFTH technique in covering the framework class associations. The number of test cases generated using the TFTH technique is less than half the number of test cases generated using the New Framework Test approach. However, most of the useful test cases (85.3%) generated using the TFTH technique are not covered using the other testing approach. These test cases are generated to test the hooks, polymorphic behaviors, inheritance of the framework objects, associations between framework and interface classes, and framework extensibility.

Since the New Framework Test approach does not consider testing the extensibility of the framework, none of the generated test cases are useful to perform this important framework testing aspect. On the other hand, more than one-third of the test cases generated using the TFTH technique test the extensibility of the framework. These test cases can be augmented at the framework instantiation testing process.

Finally, the results given in Table II show that the TFTH technique focuses the testing on one fourth of the classes covered by the New Framework Test approach. This indicates that focusing on a few of the framework classes (i.e., focusing on the anticipated to be used classes) is enough to cover most of the useful test cases.

B. Generating Test Cases for the WaveFront Pattern Framework

The WaveFront Pattern (WFP) [13] is a pattern that supports the computation of dependent elements. The pattern is used to generate frameworks automatically using the CO_2P_3S parallel programming system [22]. A generated WFP framework is considered in this experiment. The framework is relatively small, consisting of six classes and about 150 lines of code. Three hooks are used to document how to use the framework.

Both the New Framework Test approach and the TFTH technique are applied to generate test cases for the WFP framework. The results of applying the testing techniques on the framework are shown in Table III. These results are generally close to the ones shown in Table II.

TABLE II
RESULTS OF APPLYING THE NEW FRAMEWORK TEST APPROACH AND TFTH TECHNIQUE ON CSF FRAMEWORK

	New Framework Test approach				TFTH technique
	N+ Test	Extended Use Case Test	Class Association Test	Total	
Number of generated test cases	1067	63	132	1262	614
Number of useful test cases	244	63	57	364	614
Usefulness degree	0.227	1.0	0.432	0.288	1.0
Uniqueness degree	-	-	-	0.902	1.0
Number of classes/objects focused on (out of 40 classes)	40	10	40	40 (100% of the CSF classes)	10 (25% of the CSF classes)
Number of useful test cases not covered by the other testing technique	1 (0.4% of the useful test cases)	0 (0% of the useful test cases)	7 (12.3% of the useful test cases)	8 (2.2% of the useful test cases)	524 (85.3% of the useful test cases)
Number of test cases to test the extensibility of the framework	0	0	0	0 (0% of the generated test cases)	232 (37.8% of the generated test cases)

TABLE III
RESULTS OF APPLYING THE NEW FRAMEWORK TEST APPROACH AND TFTH TECHNIQUE ON WAVEFRONT PATTERN FRAMEWORK

	New Framework Test approach				TFTH technique
	N+ Test	Extended Use Case Test	Class Association Test	Total	
Number of generated test cases	128	23	35	186	125
Number of useful test cases	61	23	13	97	125
Usefulness degree	0.477	1.0	0.371	0.522	1.0
Uniqueness degree	-	-	-	0.691	1.0
Number of classes/objects focused on (out of 6 classes)	6	2	6	6 (100% of the WFP framework classes)	2 (33.3% of the WFP framework classes)
Number of useful test cases not covered by the other testing technique	0 (0% of the useful test cases)	0 (0% of the useful test cases)	1 (7.7% of the useful test cases)	1 (1.0% of the useful test cases)	95 (76% of the useful test cases)
Number of test cases to test the extensibility of the framework	0	0	0	0 (0% of the generated test cases)	65 (52% of the generated test cases)

VI. CONCLUSIONS AND FUTURE WORK

In this paper, two measures are introduced to evaluate the adequacy of two object-oriented framework system-based testing techniques. The two measures are usefulness and uniqueness of the testing techniques. The two testing techniques considered in the comparison study are the New Framework Test approach and the TFTH technique. The two testing techniques are also analytically compared. The analytical comparison shows that the TFTH technique is more powerful in covering the class associations between the instantiation and framework classes, the hooks, the framework component inheritance, the polymorphic behaviors, and framework extensibility. The New Framework Test approach is more powerful in covering the framework class-state transitions, the associations among the framework classes, and the inheritance behaviors among the framework components. Two frameworks are used in the experimental comparison study: the CSF and the WFP. The experimental comparison study results show that the number of test cases generated using the TFTH technique is less than the number of test cases generated using the New Framework Test approach. However, the TFTH technique is better than the New Framework Test approach in terms of usefulness and uniqueness measured values. The TFTH technique can be enhanced by combining it to the Class Association Test to cover the framework class associations.

In our future research, we plan to compare the two testing techniques in terms of fault coverage. In addition, we plan to apply the introduced usefulness and uniqueness measures to compare testing techniques introduced for testing types of software other than object-oriented frameworks.

ACKNOWLEDGMENT

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

REFERENCES

[1] H. Chen, T. Tse, and T. Chen, Jan. 2001, TACCLE: a methodology for object-oriented software Testing At the Class and Cluster Levels, *ACM Transactions on Software Engineering and Methodology*, Vol.10, No.1, pp.56-109.

[2] K. Beck and R, Johnson, 1994. Patterns generated architectures, *Proc. of ECOOP 94*, 139-149.

[3] G. Froehlich, 2002. Hooks: an aid to the reuse of object-oriented frameworks, *Ph.D. Thesis, University of Alberta, Department of Computing Science*.

[4] R. Binder, 1999. *Testing object-oriented systems*, Addison Wesley.

[5] W. Tsai, Y. Tu, W. Shao, and E. Ebner, October, 1999. Testing extensible design patterns in object-oriented frameworks through scenario templates, *23rd Annual International Computer Software and Applications Conference, Phoenix, Arizona*.

[6] Y. Wang, D. Patel, G. King, I. Court, G. Staples, M. Ross, and M. Fayad, March 2000, On built-in test reuse in object-oriented framework design, *ACM Computing Surveys (CSUR)*, Vol. 32(1es), pp. 7-12.

[7] J. Al Dallal and P. Sorenson, September 2002, System testing for object-oriented frameworks using hook technology, *Proc. of the 17th IEEE International Conference on Automated Software Applications (ASE'02)*, Edinburgh, UK, pp. 231-236.

[8] J. Al Dallal and P. Sorenson, 2005, Reusing class-based test cases for testing object-oriented framework interface classes, *Journal of Software Maintenance and Evolution: Research and Practise*, Vol. 17, No. 3, pp. 169-196.

[9] 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.

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

[11] 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.

[12] J. Al Dallal, Testing object-oriented hook methods, accepted for publication in the *Kuwait Journal of Science and Engineering*, 2007.

[13] 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. April 2002

[14] R. Binder, Testing object-oriented software: A survey, *Software Testing, Verification and Reliability*, Vol. 6 No. 3/4, pp. 125-252, 1996.

[15] 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*, November 2006, Vol. 16, pp. 90-95.

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

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

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

[19] A. Tevanlinna, J. Taina, and R. Kauppinen, Product family testing: a survey, *ACM SIGSOFT Software Engineering Notes*, Vol. 29, No. 2, pp. 12-18, 2004.

[20] J. McGregor, Testing a software product line, *Technical Report CMU/SEI-2001-TR-022,* Software Engineering Institute, Pittsburgh, PA. 2001.

[21] 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.

[22] S. McDonald, J. Schaeffer, and D. Szafron, Pattern-based object-oriented parallel programming. *Proceedings of the First International Scientific Computing in Object-Oriented Parallel Environments Conference (ISCOPE'97)*, Vol. 1343 of Lecture Notes in Computer Science, 1997, pp 167-274.

Jehad Al Dallal received his B.Sc. and M.Sc. in degrees in computer engineering from Kuwait University in Kuwait in 1995 and 1997, respectively. He received his PhD degree in computer science from the University of Alberta in Canada in 2003.

He is currently working at Kuwait University, Department of Information Sciences, as an assistant professor. His research interests include software testing and software analysis.