Estimating the Coverage of the Framework Application
Reusable Cluster-Based Test Cases

Jehad Al Dallal  
Department of Information Sciences  
Kuwait University  
P.O. Box 5969, Safat 13060, Kuwait  
jehad@cfw.kuniv.edu

Paul Sorenson  
Department of Computing Science  
University of Alberta  
Edmonton, AB. T6G 2H1  
sorenson@cs.ualberta.ca

Abstract
Object-oriented frameworks support both software code and design reusability. In addition, it is found that providing class-based tests with the framework reduces considerably the class-based testing time and effort of the applications developed using the frameworks. Similarly, reusable cluster-based test cases can be generated using the framework hooks, and they, too, can be provided with the framework to reduce the cluster testing time and effort of the framework applications.

In this paper, we introduce a methodology to estimate the possible coverage of the cluster-based reusable test cases for framework applications prior to suggesting and applying a specific technique to produce the test cases. An experimental case study is conducted to demonstrate the practical issues in applying the introduced methodology and to give insights on the possible coverage of the framework reusable cluster-based test cases. The results of applying the methodology on five framework applications show that, on average, the reusable cluster-based test cases cover at least one-third of the cluster-testing areas of the interface classes created during the framework application engineering stage.

Key Words  
case study, cluster testing, framework interface classes, hooks, object-oriented framework, object-oriented framework application, reusable test cases.
1. Introduction

A popular goal of software engineering is to develop techniques and tools to assist in design and implementation to meet the market requirements. Meeting time-to-market demands for a software product or application is often vital to the success of an organization or project. In the highly competitive software market, customers seem to demand less time for development while simultaneously expecting better products. Using object-oriented frameworks is an appealing way to speed up the development of the software product family [1]. A software product family is a set of software products that share common features [2]. An application framework is the reusable design and implementation of a system or subsystem [3]. Application developers can complete or extend a suitable framework to build their particular applications instead of developing the applications from scratch. Design for reusability is a costly and time-consuming task. However, reusing the framework design and code reduces the application development time and cost considerably. As a result, there exists a high probability that the original investment will be recouped after producing a few framework applications.

Researchers commonly limit framework reusability to only code and design, which forces the application developers to spend considerable time and effort in testing their applications from scratch. Software testing is a critical and important stage of the application software development life cycle that affects the overall software quality. In a typical programming project, approximately half of the effort is spent on testing activities (i.e., validation and verification) [4]. Therefore, extending the framework reusability to test artifacts potentially can reduce the framework application testing time and increase application quality. Providing the frameworks with reusable test cases makes the frameworks more appealing and encourages application developers to use the frameworks.

Generally, to test an object-oriented application, four main testing levels have to be exercised, including method testing, class testing, cluster testing, and system testing [5]. 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 on the basis of acceptance testing requirements. The basic element in testing is a test case, which typically consists of a set of inputs, execution conditions, and expected results. Reusable test cases are the ones created once and reused frequently across different applications or different versions of the same application. To build reusable test cases, it is required to identify the commonalities among a set of applications. There are two types of commonalities among a set of applications: (1) the reusable assets used to build the applications and (2) the steps followed to use the reusable assets. To use the reusable test cases for testing a specific application, a mechanism is required to identify the applicable test cases. For the applicable test cases, a technique is required to identify whether the test cases can be used as-is or they have to be modified or extended. If the test cases have to be modified or extended, the way to perform such modification or extension should be introduced previously. Therefore, the reusable test cases have to be designed carefully in such a way that they become effective and easy to use in testing the applications. The cost of reusing the test cases must be much lower than the cost of building the test cases from scratch; otherwise, application developers will prefer to build their own test cases.

The reusability of test cases is shown to be useful in reducing the testing cost in several testing areas, including regression testing (e.g., [6,7,8,9,10]), testing subclasses (e.g., [11]), testing the use of class libraries (e.g., [12]), testing software product-lines [13,14,15]), and testing object-oriented framework applications [16,17,18,19]. Building reusable test cases for the framework applications at the framework development stage increases the framework development time and cost. We believe that the original investment cannot be fully realized unless the reusable test cases are effective and easy to use in testing the applications. Generally, the effectiveness of the test cases depends on their error detection power. The error detection power depends on the coverage of the test cases for the error-prone pieces of code. The nature of these pieces of code varies from one testing level to another. For example, the class-level error-prone pieces of code are those causing intraclass and super/subclass interactions. The cluster-level error-prone pieces of code are those causing collaborations and interactions between the system
classes. Therefore, studying the effectiveness of the cluster-based test cases implies studying the coverage of the test cases for the pieces of code that cause collaborations and interactions between the system classes. Measuring the test case coverage gives important insights into the adequacy of testing.

Introducing a reusable test-case-generation technique and applying it are time- and cost-consuming tasks and cannot be invested unless the effectiveness of the reusable test cases in testing the targeted software system is high or at least satisfactory. Therefore, it is important to study the possible coverage of the anticipated reusable test cases prior to introducing and applying a specific reusable test-case-generation technique.

In this paper, we introduce a novel methodology to estimate the possible coverage of the framework reusable cluster-based test cases prior to suggesting and applying a specific technique for building the test cases. In addition, we conduct a case study that explains how to apply the methodology to estimate the coverage of the possible cluster-based reusable test cases for framework applications. The case study uses five applications developed using a Client-Server Framework (CSF). The case study also evaluates experimentally the factors that affect the coverage results. The coverage of the reusable test cases is measured in terms of the covered Framework Interface Class (FIC) code statements that cause class collaborations and interactions. FICs are application-specific classes that extend or use the framework classes directly [17]. The results of the case study show that, on average, more than one-third of the FIC class interactions are covered using the possible reusable cluster-based test cases. This result indicates that building and providing with the framework reusable cluster-based test cases is expected to reasonably reduce the cluster testing efforts at the framework application development stage. The results also show that applying the introduced estimation methodology requires relatively exhaustive work. However, this work is performed once for a certain type of reusable code and can be recouped when applying reusable cluster-based test cases for testing applications that use this type of reusable code. Finally, the results of the case study show that following properly the instructions for building the framework applications increases the coverage of the reusable cluster-based test cases and consequently decreases the testing time and effort consumed during the framework application development stage.
The paper is organized as follows. Section 2 overviews the concepts in the object-oriented framework paradigm and discusses the related work in the area of test case reusability. Section 3 introduces a methodology to estimate the possible coverage of the framework reusable cluster-based test cases. A case study in which the methodology is applied is explained in Section 4. Finally, Section 5 provides a conclusion and discussion of future work.

2. Background and Related Work

This paper addresses the reusability of the object-oriented cluster-based test cases. In this section, the research work in the area of object-oriented framework paradigm and the area of test case reusability are overviewed.

2.1. Object-oriented framework paradigm

An application framework provides a reusable design and implementation for a family of software systems [3]. It contains a collection of reusable concrete and abstract classes. The framework design provides the context in which the classes are used. The framework itself is not complete. Users of the framework complete or extend the framework to build their particular applications. Places at which users can add their own classes are called hook points [20,21].

Object-oriented framework engineering is divided into separate domain and application engineering tasks. During domain engineering, the framework assets are produced. During application engineering, the users of the framework complete or extend the framework assets to build their particular instantiations. When the framework is used during the application engineering stage, developers build two types of classes: (1) classes that use the framework classes and (2) classes that do not. Classes that use the framework classes are called Framework Interface Classes (FICs) [17] because they act as interfaces between the framework classes and the second type of the classes created by application developers. Figure 1 shows the relationship between the framework classes, the hook points, the FICs, and the other application classes.
Figure 1: Framework application classes

There are three areas related to the FICs for which cluster testing can be applied: (1) the interactions between the FICs and the framework classes, (2) the interactions among the FICs, and (3) the interactions between the FICs and the other application classes.

In [20,21], the issue of documenting the purpose of a framework and how it is intended to be used using the hooks is described and formalized. The hook description concept is introduced. The hook description includes the implementation steps and the specifications (i.e., pre-conditions and post-conditions) of the FIC methods. Froehlich [20] provides 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 preconditions (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 done to develop the application, (8) the post conditions (i.e., constraints on the parameters that must be true after the hook has been used), and (9) a general comment section. It is not necessary to have all of the above parts for each hook. Froehlich [20] identifies four levels of support provided for adaptation within the framework: option, supported template, open, and evolutionary. At the option level, a number of pre-built components are provided within the framework, and the developer chooses one without requiring extensive knowledge about the framework. At the template level, the developer supplies parameters for components and follows a well-supported pattern of behavior. At the open level, the developer adds new properties to classes, new classes to the framework, or extends the
framework’s functionality. At the evolutionary level, the developer changes parts of the framework code or breaks invariants defined on the framework. The four levels of support are provided at the hook points. A hook description documents the use of the framework at the first three levels of support. The fourth level of support describes the framework misuse, and therefore, it is not documented.

Figure 2 shows two hook description examples for a creation of an object needed to communicate across the network and a creation of a mail server object. When the New CommAwareObject hook is used at the development process of an application, the first statement in the Changes section of the hook description is used to create a class that extends the framework class CommAwareObject. The second statement is used to create a method init in the created class. Finally, the third statement calls the Choose MailServer hook. The first statement in the Changes section of the Choose MailServer hook description provides three options for the required mail server. The application developer has to select one of the three pre-built components in the framework. The second and third statements are used for specifying the values of the IP and port of the required mail server. Finally, the fourth statement is used for the creation of a mail server object in the init method of the class created using the first statement in the Changes section of the New CommAwareObject hook.

Figure 3 shows an example of sample Java code created using the hook descriptions given in Figure 2. In this sample code, a subclass MyCOA that extends the CommAwareObject class is created. The subclass includes the init method. According to the application developer selection, a MailServer object is created inside the body of the init method. The parameters of the constructor method of the MailServer class are assigned to values given by the application developer.
2.2. Test Case Reusability

The testing areas for which reusability of test cases for object-oriented software are proposed and discussed include regression testing, testing subclasses, testing the use of class libraries, testing software product-lines, and testing object-oriented framework applications.

In regression testing [6,7,8,9,10], a modified version of the software is tested to provide confidence that the changed parts are behaving as intended and that the unchanged parts are not affected by the modifications in an unforeseen way. The test

---

**Figure 2**: Two CSF hook description examples

**Name: New CommAwareObject**

Requirement: An object needs to communicate across the network.

... Changes:
- New subclass NewCAO of CommAwareObject;
- New operation NewCAO.init();
- New Choose Mail Server [NewCAO=NewCAO]

**Name: Choose Mail Server**

Requirement: Choose a particular type of mailserver to suit the application.

... Changes:
- Choose MS from (MailServer, AppletMailServer, AppletServerMailServer);
- Acquire IP : string;
- Acquire Port : integer;
- Create Object MServer as MS(IP,Port) in NewCAO.init();

... 

**Figure 3**: Java sample code created using the hook descriptions given in Figure 2

```java
Class MyCAO extends CommAwareObject {
    public void init() {
        MailServer   MyMS=new MailServer("foo",34356);
    }
}
```

---

Class MyCAO extends CommAwareObject {
    public void init() {
        MailServer   MyMS=new MailServer("foo",34356);
    }
}
suite used to test the original version of the software or part of it is reused to test the modified version. In subclass testing [11], the superclass test suite or part of it has to be reapplied to gain confidence that the inherited superclass features work correctly in the context of the subclass. In testing the use of the class libraries and the frameworks, Binder [12] states that the class library user and the framework user can reuse the class libraries test suite and the framework test suite, respectively, at the cluster-testing level without introducing new specific approaches. Tevanlinna et al. [1] identifies four different strategies to model product family testing. The first strategy is to test product by product without considering the benefits of reuse (e.g., [22]). The second strategy tests the first product individually and tests the following products incrementally using regression testing techniques. The third strategy builds reusable test cases extensively during the domain engineering stage. These test cases are reused as-is or customized during the application engineering stage to test the product-specific aspects [14]. In addition, each time that a product is constructed and tested, the specialized test cases and the new test cases applied for testing the product are stored in the database. The fourth strategy applies unit testing in the domain engineering stage and integration, system, and acceptance testing at the application engineering stage. In [23], the suggestion is made to put built-in tests inside the reusable components and to build component testers. The built-in-tests are in the form of methods that provide information to test the component. The component testers exercise the built-in tests to test the component. Whenever the components are used, the component testers can be used as-is or modified to verify that the components work correctly in their deployment environment.

Several techniques are introduced to test object-oriented frameworks and their instantiations. Several recent research studies address the problem of object-oriented testing at different levels in general. Some of these testing techniques are proposed specifically to test object-oriented frameworks, including [1,12,24,25,26]. Others are proposed specifically to test object-oriented framework instantiations, including [16,17,18,24,27].

Tsai et al. [27] discuss the issues of testing instantiations developed with design patterns using object-oriented frameworks. They have introduced 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 interaction between framework objects and custom objects. Wang et al. [16], propose providing the framework with reusable class-based test cases that can be applied during the application engineering stage. Al Dallal [17], and Al Dallal and Sorenson [18,19] propose a technique to test the FICs at class level using reusable test cases built during the framework domain engineering stage.

In conclusion, only Tsai et al. [27] consider building reusable cluster-based test cases for framework instantiations. However, Tsai et al. did not study the coverage of these test cases when they are used. Except for [17] and [28], none of the research reviewed in the areas of regression testing, testing subclasses, testing the use of class libraries, testing software product-lines, and testing object-oriented framework applications includes an empirical study of the possible coverage of the reusable test cases built using the introduced techniques. In [17], two case studies are conducted to examine the coverage of the reusable class-based test cases in testing the FICs. The coverage is measured in terms of the number of covered transitions in the state-transition models of the FICs. The case studies are conducted using five applications developed using Client-Server Framework and six applications developed using SalesPoint framework, respectively. The results of the two case studies show that the reusable class-based test cases cover, on average, 77% and 96%, respectively, of the transitions in the specification models of the FICs. These two case studies are conducted to examine the coverage of a specific technique that builds reusable test cases, and therefore, no conclusion is reached concerning the coverage of the reusable test cases regardless of the applied test-case-generation technique. In addition, the case studies are limited for class-level testing.

In [28], three case studies are conducted to examine the coverage of the reusable class-based test cases in testing the application-specific classes built during the application engineering stage. The coverage is measured in terms of the percentage of the number of application-specific classes for which reusable test cases can be built to the total number of classes built during the application engineering stage. The three case studies are conducted using five applications developed using Client-Server Framework, six applications developed using the SalesPoint framework, and ten applications developed using Swing Java framework, respectively. The results of the three case
studies show that the reusable class-based test cases cover, on average, 41%, 69%, and 15%, respectively, of the classes built during the application engineering stage. The results of the case studies indicate that it is worthwhile to build class-based reusable test cases for applications developed using domain-specific frameworks, as the original investment will be recouped after producing a few number of framework applications. However, for some frameworks used in different application domains, it might not be worthwhile to build class-based reusable test cases. The case studies are conducted to examine the possible coverage of the reusable class-based test cases regardless of the testing technique applied for building the test cases. In this paper, we study the possible coverage of the reusable cluster-based test cases for framework instantiations, regardless of the testing technique applied for building the test cases. The work in this paper is more sophisticated than the work applied in [28] because, in [28], the case studies are conducted by counting the number of FICs developed in the instantiations without inspecting the source code of the FICs. In this paper, the estimation of the possible coverage of the reusable cluster-based test cases requires inspecting the source code of the FICs line by line.

3. Estimating the possible coverage of the reusable cluster-based test cases

Actual measuring for the reusable cluster-based test case coverage is performed by (1) executing the test cases, (2) counting the number of code statements that cause class interactions and are covered by the test cases, (3) counting the total number of code statements that cause class interactions, and (4) calculating the percentage of the number of code statements counted in Step 2 to the number of code statements counted in Step 3. Introducing a technique to generate reusable cluster-based test cases, applying the technique to build the test cases, and applying the test cases to perform the actual testing are costly and time-consuming tasks. After spending this time and cost, it would be disappointing to discover that the coverage of the reusable test cases is relatively low, and therefore, the invested time and cost are not recouped when using the reusable test cases. To solve this problem, we introduce a methodology to estimate the possible coverage of the reusable cluster-based test cases prior to introducing the test-case-generation technique and building and applying the reusable test cases. In this paper, the
methodology is introduced and applied to estimate the coverage of the reusable cluster-based test cases for framework applications. The methodology includes four steps as follows:

**Step 1: Detect the hook description statements that cause class interactions.** In [29] and [30], several types of code statements that cause class interactions are identified, including message passing (i.e., invoking a method of a class in another class), data abstraction (i.e., an attribute of a class that has another class as its type), and parameter passing (i.e., passing an object as a parameter value). The implementation steps included in the *Changes* section of the hook description are used to build code statements during the application engineering stage. Some of these statements cause class interactions; they define the possible interactions between the FICs and the framework classes and the interactions among the FICs. For example, when the bold statement in the *Changes* section of the Choose MailServer hook description given in Figure 2 is activated during the application engineering stage, it produces a code statement that defines an interaction between the class that extends the CommAwareObject and the selected mail server class. An instance of this code statement is shown in the *init* method block given in Figure 3. The *Changes* section of the hook description can include implementation steps used to generate any type of code statements causing class interactions. For example, the bold implementation step given in Figure 2 is used to generate a code statement that invokes the constructor method of a mail server (i.e., MailServer, AppletMailServer, or AppletServerMailServer) in the subclass of CommAwareObject class. In addition, the same implementation step generates a code statement in which a string object is passed as a parameter value. Finally, the *Changes* section can include implementation steps used to generate code statements in which an attribute of a class has another class as its type. As a result, hook descriptions include valuable information that can be used as bases to build reusable cluster-based test cases during the framework domain engineering stage. It is assumed that any reusable cluster-based test case generating technique should produce test cases to cover each anticipated code statement built using the hook descriptions and cause class interactions at least once. In this methodology step, all the *Changes* sections
of all hook descriptions have to be inspected to identify the possible resulting code statements that cause class interactions.

**Step 2: Detect and count the application-specific code statements that cause class interactions.** In this step, the code statements of the implemented FICs are inspected. Any code statement that causes class interaction is identified and counted. Figure 4 provides a sample code listing for the clientControlDataHandler FIC of the Student management system application built using Client-Server Framework. The code statements that cause class interactions are identified and bolded. For example, the code statement number 8 is bold because it causes an interaction between the clientControlDataHandler class and the Message and CommAwareObject classes as shown in the passed parameters of the called method. The number of bold code statements is 17.

```
1. class clientControlDataHandler extends MessageHandler {
2.   private Applet A;
3.   public clientControlDataHandler() {}{
4.   public clientControlDataHandler(Applet appletobject) {
5.     A = appletobject;
6.   }
7.   public void handleMessage(Message m, CommAwareObject coa) {
8.     super.handleMessage(m, coa);
9.     System.out.println("Message <" + m.getType() + "> received.");
10.    instrControlData icd = new instrControlData();
11.    icd = (instrControlData)m.getData();
12.    List tempTeam = new List();
13.    tempTeam = icd.getLoginList();
14.    List tempPass = new List();
15.    tempPass = icd.getPasswordList();
16.    int tcount = tempTeam.getItemCount();
17.    int pcount = tempPass.getItemCount();
18.    for (int n = 0; n < tcount; n ++) {
19.      String temp = tempTeam.getItem(n);
20.      ((editTeam)A).tList.add(tempTeam.getItem(n));
21.    }
22.    for (int n = 0; n < pcount; n ++) {
23.      String temp = tempPass.getItem(n);
24.      ((editTeam)A).pList.add(tempPass.getItem(n));
25.    }
26. }
27. }
```

**Figure 4**: Java code of the clientControlDataHandler FIC
Step 3: Detect the application-specific code statements that are generated using the hook descriptions and cause class interactions. This step identifies the code statements identified in Step 2 and generated using hook description statements identified in Step 1. These statements are italicized in the sample code listing given in Figure 4. There are two instances of such statements given in lines 7 and 8. These statements are generated using the hook descriptions of the Client-Server framework.

Step 4: Calculate the estimated range of possible coverage of the reusable cluster-based test cases. When the reusable cluster-based test cases are executed, all statements identified in Step 3 are exercised according to the assumption given in Step 1. Consequently, all the code statements that have the same control flow as the statements identified in Step 3 are also exercised. Some of these statements are not created using hook descriptions and they cause class interactions. As a result, these statements are luckily covered despite the fact that the reusable test cases were not originally built to cover them. In the sample code listing given in Figure 4, the code statements 9-17 have the same flow of control that includes statements 7 and 8, and therefore, they are also covered using the reusable test cases. The code statements not included in the flow of control of the code statements identified in Step 3 may or may not be covered by the reusable test cases. This depends on the control values that have to be satisfied to execute these statements. For example, in the sample code listing given in Figure 4, if luckily the code exercised by the reusable test cases assigns values greater than zero to the `tcount` and `pcount` variables, the statements 19, 20, 23, and 24 will be covered by the reusable test cases; otherwise, these statements are not covered. As a result, for this example, the estimated possible coverage of the reusable test cases varies between 11 (i.e., statements 7-17) and 15 statements (i.e., statements 7-17, 19, 20, 23, and 24) out of the 17 code statements identified in Step 2. The percentage of the estimated possible coverage of the reusable test cases for `clientControlDataHandler` FIC ranges between 64.7-88.2%.

The difference between the minimum and maximum estimated possible coverage depends in some cases, as in the given example, on the application under test. In other cases, the difference depends on the cluster-based testing technique that can be introduced to build the reusable test cases. In cluster-based testing, sequences of method
calls are exercised. The values assigned to the parameters of the methods are decided according to the applied testing technique. In some testing techniques, the same method is exercised several times with different values assigned to the method parameters. For example, assume that the values of the \( tcount \) and \( pcount \) variables given in the code shown in Figure 4 depend on the parameters passed to the \( \text{handleMessage} \) method, and the applied cluster-based testing technique assigns parameter values that cause the \( tcount \) and \( pcount \) variables to have values greater than zero. In this case, the statements 19, 20, 23, and 24 will be covered by the reusable test cases; otherwise, these statements are not covered.

4. Case study

The purpose of this case study is to explain experimentally how to use the introduced methodology to study the possible coverage of the framework reusable cluster-based test cases. In addition, the case study aims to demonstrate the amount of work involved in estimating the possible coverage of the reusable test cases. Five applications developed using one framework were considered in this case study. This section introduces the framework, illustrates the case study settings, and reports the coverage results and the factors that affect the coverage of the reusable cluster-based test cases.

4.1. Used framework

Five applications developed using Client-Server Framework (CSF) [20] were considered in the case study. 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. CSF also provides persistent storage capabilities and can handle the communications over a TCP/IP connection using a model similar to email. 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 (LOC) (without comments/blank lines). The CSF has 16 hook descriptions that include the implementation steps required to build 34 hook methods for ten FICs and provide their specifications.
4.2. Case study settings

In the first step of the case study, the Changes sections of the CSF sixteen hook descriptions were inspected. In this inspection, the implementation steps that result in building code statements that cause class interactions are detected. Thirty-one instances of such implementation steps were detected out of a total of 97 implementation steps included in all the CSF hook descriptions.

In the second step, the code statements of the implemented FICs in the framework applications are inspected to detect the statements that cause class interactions. Performing this inspection required in this step for a relatively large number of applications in the absence of a tool support requires exhaustive effort. Therefore, the case study was conducted using five selected applications out of a pool of 15 CSF applications. The total number of LOC inspected manually to obtain the results is approximately 8K, not including comments and blank lines. The selected applications were considered previously [17] in studying the reusability of the class-based test cases. The applications were developed by students as part of a fourth-year undergraduate course project in software engineering at the University of Alberta.

In the third step, the code statements that cause class interactions and are detected in the second step are analyzed manually. In this analysis, the statements that were generated using the hooks are detected and counted. The detection is performed manually by comparing the code statements to the code statements that can be generated using the implementation steps documented in the Changes sections of the hook descriptions.

In the fourth step, the control flow of each method in each implemented FIC in each of the considered applications is analyzed manually. During the analysis, we have counted the number of code statements detected in Step 2 that have the same flow of control of the code statements detected in Step 3. We assume that any reusable cluster-based test case generating technique should cover at least the code statements detected in Step 3 because these statements are generated using the knowledge known during the domain engineering stage as discussed in Section 3. Therefore, these test cases, when executed, cover also all of the code statements detected in Step 4 because they have the same flow of control. As a result, the number \( n \) of statements detected in this step is the
minimum number of code statements causing class interactions and covered by reusable cluster-based test cases generated using any technique satisfying the given assumption. After that, we counted manually the number $m$ of code statements causing class interactions, included in the methods containing code statements detected in Step 3 and not included in the flow of control of the code statements detected in Step 3. These code statements may or may not be covered by the reusable cluster-based test cases. This depends greatly on the applied technique for generating the reusable cluster-based test cases, which is undetermined during the time when the case study is conducted. If luckily all of these code statements are covered, the maximum number of code statements causing class interactions and can be covered by the reusable cluster-based test cases will be equal to the summation of $n$ and $m$.

The set of code statements causing class interactions and not included in any method containing code statements detected in Step 3 cannot be covered by the reusable cluster-based test cases. The reason is that the methods that include the set of code statements are unknown during the domain engineering stage.

4.3. Case study results

For each application, the classes implemented during the application engineering stage were counted. The number of classes does not include the number of framework classes. The number of classes was counted using the LOCC tool [31]. LOCC is a Java tool that produces size data corresponding to the number of packages, the number of classes in each package, the number of methods in each class, and the number of LOC in each package, class, and method. The first, second, and third rows of Table 1 show the application name, the total number of application classes, not including framework classes, and the number of LOC for these classes. The FICs included in the applications were counted manually and reported in the fourth row of Table 1. In addition, the fourth row reports the percentage of the number of FICs given in the fourth row to the total number of classes reported in the second row. The number of LOC of the FICs is counted using an LOCC tool and reported in the fifth row of Table 1 along with the percentage of the number of LOC to the total number of LOC reported in the third row. For each FIC in each application, the number of code statements that cause class interactions is counted
manually and reported in the sixth row of Table 1. The number of code statements that cause class interactions and are generated using the hook descriptions is shown in the seventh row of Table 1. In addition, the seventh row reports the percentage of the number of statements reported in the seventh row to the number of statements reported in the sixth row of Table 1.

The minimum and maximum numbers of code statements that cause class interactions and are covered in the reusable cluster-based test cases are shown in the eighth and ninth rows of Table 1, respectively. In addition, the eighth and ninth rows report the percentages of these numbers of statements to the number of statements reported in the sixth row of Table 1. Finally, for comparison purposes, the reusability results of the reusable class-based test cases for the same applications found in [17] are shown in the tenth row of Table 1. The figures reported in this row are for the percentages of the number of transitions covered by the reusable class-based test cases to the total number of transitions in the specification models of the FICs. The last three rows of Table 1 include detailed results showing why some of the code statements causing class interactions cannot be covered by reusable cluster-based test cases. These results are discussed at the end of this section. The last column of Table 1 summarizes the results.

For the selected applications developed using CSF framework, Table 1 (row eight) shows that the minimum number of code statements that cause class interactions and are covered in the reusable cluster-based test cases varies from 17% to 70% with an average of 38%. The maximum number of code statements that cause class interactions and are covered in the reusable cluster-based test cases varies from 25% to 82% with an average of 54% (row nine). Finally, for all of the considered CSF applications, the coverage results of the reusable class-based test cases are better (and in some cases much better) than the coverage results of the reusable cluster-based test cases. This gives an indication that the framework hooks are more useful for building reusable class-based test cases than for building reusable cluster-based test cases.
<table>
<thead>
<tr>
<th>Application name</th>
<th>Student Management System</th>
<th>Chatting System</th>
<th>Course Management System</th>
<th>StoneClash Strategy Game</th>
<th>Army Game</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of classes</td>
<td>47</td>
<td>55</td>
<td>44</td>
<td>106</td>
<td>149</td>
<td>80</td>
</tr>
<tr>
<td>Number of LOC</td>
<td>3887</td>
<td>7464</td>
<td>3191</td>
<td>5324</td>
<td>8792</td>
<td>5732</td>
</tr>
<tr>
<td>Number of FICs</td>
<td>31 (66%)</td>
<td>3 (5.5%)</td>
<td>17 (38.6%)</td>
<td>56 (52.8%)</td>
<td>66 (44.3%)</td>
<td>35 (43.7%)</td>
</tr>
<tr>
<td>Number of LOC of the FICs</td>
<td>1586 (40.3%)</td>
<td>179 (2.4%)</td>
<td>669 (20.9%)</td>
<td>2050 (38.5%)</td>
<td>3449 (39.2%)</td>
<td>1587 (27.7%)</td>
</tr>
<tr>
<td>Number of code statements that cause class interactions</td>
<td>882</td>
<td>106</td>
<td>291</td>
<td>884</td>
<td>1426</td>
<td>718</td>
</tr>
<tr>
<td>Number of statements created using the hook descriptions and cause class interactions</td>
<td>79 (9%)</td>
<td>21 (20%)</td>
<td>41 (14%)</td>
<td>95 (11%)</td>
<td>158 (11%)</td>
<td>79 (13%)</td>
</tr>
<tr>
<td>Minimum number of statements that cause class interactions and can be covered by reusable cluster-based test cases</td>
<td>613 (70%)</td>
<td>32 (30%)</td>
<td>120 (41%)</td>
<td>295 (33%)</td>
<td>236 (17%)</td>
<td>259 (38%)</td>
</tr>
<tr>
<td>Maximum number of statements that cause class interactions and can be covered by reusable cluster-based test cases</td>
<td>723 (82%)</td>
<td>55 (52%)</td>
<td>166 (57%)</td>
<td>498 (56%)</td>
<td>363 (25%)</td>
<td>361 (54%)</td>
</tr>
<tr>
<td>Class-based test cases coverage results</td>
<td>89%</td>
<td>60%</td>
<td>71%</td>
<td>77%</td>
<td>77%</td>
<td>77%</td>
</tr>
<tr>
<td>Number of statements that cause class interactions and are not covered because of defining interactions with other application classes</td>
<td>40 (4.5%)</td>
<td>42 (39.6%)</td>
<td>74 (25%)</td>
<td>266 (30.1%)</td>
<td>315 (22.1%)</td>
<td>147 (24.4%)</td>
</tr>
<tr>
<td>Number of statements that cause class interactions and are not covered because of misusing framework hook descriptions</td>
<td>18 (2%)</td>
<td>2 (1.9%)</td>
<td>38 (13.1%)</td>
<td>105 (11.9%)</td>
<td>748 (52%)</td>
<td>182 (16.3%)</td>
</tr>
<tr>
<td>Number of statements that cause class interactions and are not covered because of using open hooks</td>
<td>101 (11.5%)</td>
<td>7 (6.6%)</td>
<td>13 (4.5%)</td>
<td>15 (1.7%)</td>
<td>0 (0%)</td>
<td>27 (4.8%)</td>
</tr>
</tbody>
</table>

Table 1: The results of using the CSF reusable cluster-based test cases for testing CSF applications
4.4. Cluster-based test cases coverage factors

During the analysis performed in the case study, we observed the following factors that negatively affect the coverage of the reusable cluster-based test cases:

1. Defining interactions with other application classes. As discussed in Section 2, FICs can interact with three types of classes: framework classes, other FICs, and other application classes. At the application engineering stage, the application developer can add, to the FICs, code statements that cause interactions between the FICs and other application classes. These code statements are not specified in the hook descriptions, the sources of the reusable test cases. Therefore, these interactions cannot be covered by the reusable test cases.

2. Misusing the framework hook descriptions. The Changes section of the framework hook description can include implementation steps used to generate code statements causing class interactions. These statements can have reusable cluster-based test cases. When the implementation steps in the hook description are not followed properly, the test cases that cover the resulting code statements cannot be used, which reduces the coverage of the test cases. This factor greatly reduced the possible coverage of the reusable cluster-based test cases for the Army Game application as shown in the sixth column of Table 1 because the application developers misused the framework hook descriptions greatly.

3. Using open hooks. Open hooks provide neither options from which the user can select nor templates to be followed. Instead, they provide the pre- and post-conditions of the FIC methods without any code guidance. As a result, no cluster-based test cases can be built for such methods. When these methods are implemented at the application engineering stage, cluster-based test cases have to be created from scratch.

For each of the five considered applications, we have identified manually the code statements that cause class interactions and cannot be covered in the reusable cluster-based test cases. Then we have categorized these statements according to the factors that affect their coverage and counted the number of code statements in each category. The number of code statements that cause class interactions and are not covered because of (1) defining interactions with other application classes, (2) misusing the framework hook
descriptions, and (3) using open hooks are shown in the last three rows of Table 1, respectively.

For the selected applications developed using CSF framework, Table 1 (row eleven) shows that the number of code statements that cause class interactions and are not covered in the reusable cluster-based test cases because of defining interactions with other application classes varies from 4.5% to 39.6% with an average of 24.4%. The number of code statements that cause class interactions and are not covered because of misusing the framework hook descriptions vary from 1.9% to 52% with an average of 16.3% (row twelve). Finally, the number of code statements that cause class interactions and are not covered in the reusable cluster-based test cases because of using open hooks varies from 0% to 11.5% with an average of 4.8% (row thirteen).

5. Conclusions and Future Work

In this paper, we addressed the problem of studying whether it is worthwhile to build reusable cluster-based test cases during the framework domain engineering stage. We introduced a methodology that estimates experimentally the possible coverage of the reusable cluster-based test cases that can be generated using the framework hook descriptions. A case study that applies the introduced methodology is conducted using five CSF applications. The results of the case study show that, on average, the possible coverage of the reusable cluster-based test cases ranges from 38% to 54%, depending on the application under test and the testing technique used in building the test cases. The coverage results of the reusable class-based test cases are always better than the coverage results of the reusable cluster-based test cases. This gives an indication that the framework hook descriptions, the sources for building the reusable test cases, are more applicable and usable for building class-based test cases than for building cluster-based test cases. This does not mean that the framework hook descriptions should be used only for building class-based test cases. Class testing and cluster testing are two different testing levels that have to be considered when testing any object-oriented application thoroughly. Therefore, the hook descriptions can be used to build reusable test cases at both testing levels. When the test cases are applied during the application engineering stage, some of the cases may not be covered by the reusable test cases, and the
application developers have to deal with these cases separately. What is indicated by the results is that it is expected that the percentage of such uncovered cases are less at class level than at cluster level.

The results also show that using the framework hook descriptions properly at the application engineering stage increases the possible reusability of the cluster-based test cases. Finally, we have observed that the average number of code statements that cause class interactions and are not covered in the reusable cluster-based test cases because of using open hooks is proportional to the number of methods defined in the framework open hooks. The CSF hook descriptions define 34 methods. Among these methods there are just two methods (5.9%) defined in open hooks. Note that the average percentage of the number of code statements that cause class interactions and are not covered because of using open hooks in the considered applications in the case study is 4.8%.

The case study did not aim to reach a general conclusion for whether it is worthwhile to introduce a cluster-based reusable test case generating technique for framework applications in general. Such a conclusion requires considering large number of available applications developed using different types of frameworks, which requires tremendous amount of work in the absence of tool support. Instead, the case study demonstrates the way to apply the introduced methodology and shows the huge amount of work involved. Developing a supporting tool would be very helpful in considerably reducing the amount of involved work and increasing the accuracy of the results.

The work introduced in this paper is unique in the sense that, as far as we know, it is the only work considering the coverage of the reusable cluster-based test cases in any domain where code is reusable. Similar case studies can be conducted to determine whether it is worthwhile to build reusable cluster-based test cases for other types of software product families including those created in a product line or using reusable components. For any such areas, it is required to find a source for the knowledge provided during the domain engineering stage to build reusable cluster-based test cases. After that, the available set of product families can be studied to estimate the possible coverage of the reusable cluster-based test cases. If the estimated possible coverage is found to be appealing, the introduction of a test case generating technique that uses the available source of knowledge to build reusable cluster-based test cases during the
domain engineering stage will be recommended. Otherwise, it will be recommended not to waste the time and effort in introducing and applying such a cluster-based test case generating technique. Whenever such a study is conducted, it is important to consider large number of applications in a variety of domain areas. Despite the fact that the amount of involved work in such studies is huge, such a study is performed once for a certain type of reusable assets. Currently, there is a huge number of applications that use each type of reusable assets. For example, there is a huge number of available frameworks, class libraries, reusable components, and software product-lines. If, for any of these types, the study concludes that it is worthwhile to build cluster-based test cases, the huge amount of work invested in the study will be recouped as soon as the reusable test cases are applied for testing multiple applications. The benefits of the study will last as the type of the reusable assets is continually in use. Introducing similar methodologies to estimate the possible reusability of test cases at other testing levels such as class and system levels is left open for further investigation.

References


