Reusing class-based test cases for testing object-oriented framework interface classes

Jehad Al Dallal and Paul Sorenson

Department of Computing Science
University of Alberta
Edmonton, AB. T6G 2H1
Email: {jehad,sorenson}@cs.ualberta.ca

SUMMARY
An application framework provides a reusable design and implementation for a family of software systems. Frameworks are introduced to reduce the cost of a product line (i.e., family of products that share the common features) and to increase the maintainability of software products through the deployment of reliable large-scale reusable components. A key challenge with frameworks is the development, evolution and maintenance of test cases to ensure the framework operates appropriately in a given application or product. Reusable test cases increase the maintainability of the software products because an entirely new set of test cases does not have to be generated each time the framework is deployed. At the framework deployment stage, the application developers (i.e., framework users) may need the flexibility to ignore or modify part of the specification used to generate the reusable class-based test cases. This paper addresses how to deal effectively with the different modification forms such that the use of the test cases becomes easy and straightforward in testing the Framework Interface Classes (FICs) developed at the application development stage. Finally, the paper discusses the fault coverage and examines experimentally the specification coverage of the reusable test cases.

KEY WORDS: class testing, framework interface classes (FICs), object-oriented framework, reusable test cases, specification-based testing.
1. INTRODUCTION

A popular goal of software engineering is to develop techniques and tools to assist design and implementation reuse 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. Object-oriented framework technology has assisted tremendously in meeting these escalating demands by providing a reusable design and implementation for a family of software systems that share common features [1]. An object-oriented framework contains a collection of reusable concrete and abstract classes. The framework design provides the context in which the classes are used. Therefore, instead of designing and implementing the applications from scratch, developers can reuse the design and implementation of the suitable frameworks and complete or extend the frameworks to build their particular applications. However, 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. In a typical programming project, approximately half of the effort is spent on testing activities (i.e., validation and verification) [2]. Therefore, extending the framework reusability to test artifacts can potentially reduce the framework application testing time and increase application quality. Providing the frameworks with reusable test cases, makes the frameworks more usable for and marketable to application developers.

Four main testing levels are introduced for object-oriented applications including method testing, class testing, cluster testing, and system testing [3]. At the method testing level, the method responsibilities are considered. At the class testing level, the intraclass interactions and superclass/subclass interactions are considered. At the cluster testing level, the collaborations and interactions between the system classes are exercised. Finally, at the system level testing, the complete integrated system is exercised usually based on acceptance testing requirements. As shown in Figure 1, the frameworks and the applications developed using the frameworks have to be tested at the four testing levels.
This paper does not address the framework testing at all. Framework testing techniques such as [4] and [5] can be applied to test the framework. Testing object-oriented applications at method and system levels is similar to conventional program testing [3]. Addressing the cluster testing of the framework applications is considered as a future work. This paper focuses on testing the framework applications at the class testing level only.

**Figure 1 about here, please**

Figure 1. The paper framework.

We have identified two types of framework application classes built at the application development stage by application developers: (1) classes that use the framework classes and (2) classes that do not. We call the classes that use the framework classes Framework Interface Classes (FICs) because they act as interfaces between the framework classes and the second type of the classes created by application developers. Instances of FICs are called framework interface objects. FICs use the framework classes in two ways: either by subclassing them or by using them without inheritance. The former customization method is used for white-box frameworks and the latter one is used for black-box frameworks [6]. This work is concerned with testing FICs developed in either of the two customization methods. Froehlich [7] developed the concept of hooks to show how the framework is used. Hooks define how to use the framework and, therefore, they define the FICs and specify the preconditions and postconditions of the FIC methods. Froehlich [7] provided a special purpose language and grammar in which the hook description can be written. The hook description includes the implementation steps and the specifications (i.e., preconditions and postconditions) of the FIC methods. A hook description example is provided in Section 2. Hook points are the places at which the framework users (i.e., application developers) can add their own FICs using the hooks. Figure 2 shows the relation between the framework classes, the hook points, the FICs, and the other application classes.

**Figure 2 about here, please**

Figure 2. Framework application classes.

Testing the FICs achieves two main goals:
1. Increasing the confidence that the implemented methods of the FICs interact properly as described in the hook descriptions.

2. Increasing the confidence that the inherited features of the framework classes work properly in the context of the FICs that extend the framework classes.

FICs are classes built at the application development stage. The implementations of the FICs do not exist at the framework development stage, therefore, we cannot test these classes at the framework development stage. Different implementations of a FIC in different framework applications are developed using the same framework hooks. Therefore, they share common specifications provided at the framework development stage. As a result, reusable test cases generated using the common specifications can be built at the framework development stage and used as-is or customized at the application development stage to test the implementations of the FICs.

Figure 3 summarizes the FIC testing process. At the framework development stage, in the first step of the FIC testing process, the specifications of the FIC methods are used to synthesize the FIC class-based testing model as described in [8]. In the second step of the FIC testing process, the model is used to generate the reusable class-based test cases for the FIC using all paths-state technique [9]. At the framework application development stage, in the third step of the FIC testing process, the test cases are used to test the implemented FICs. Finally, in the fourth step of the FIC testing process, the specifications of the FIC methods are used to evaluate the results of the test cases. This paper focuses on showing how to use the FIC reusable class-based test cases (i.e., the third step of the FIC testing process).

**Figure 3 about here, please**

Figure 3. The FIC testing process.

When developing an application, the application developers have the flexibility to ignore or modify part of the FIC specification used to generate the reusable class-based test cases and add new specifications not covered by the reusable test cases. In this paper we have proposed easy and straightforward ways to use the reusable test cases considering the flexibility that the user has in modifying the FIC specifications. We have identified the following four problems and proposed effective solutions for each of them:

(1) How to find and discard the test cases for the ignored specifications.
(2) How to map the names of the implemented FIC methods to the names of the FIC methods introduced in the hooks and used in the reusable test cases.

(3) How to test different implementations of the same FIC method introduced in the hooks.

(4) How to deal with the flexibility that the user has in modifying the parameters of the FIC methods introduced in the hooks.

This paper focuses on showing how to reuse the test cases without modifying them. Testing new specifications added by the application developer is not considered in this paper because it requires either augmenting the test cases or creating new test cases from scratch. The paper studies the fault coverage of the test cases applied at the application development stage to test the FICs in comparison to the fault coverage of the round-trip path test cases. In the round-trip path coverage [4], the transition sequences, in the state-transition specification model, that start and end with the same state and simple paths from the starting state to the final state are covered. A simple path is a path that includes only an iteration of a loop, if a loop exists in some sequence. Finally, the paper examines experimentally the specification coverage of the reusable test cases, in terms of the number of transitions in the specification models of the implemented FICs.

The paper is organized as follows. Section 2 discusses the background research. Section 3 shows how to use the reusable test cases at the application development stage. An evaluation is reported in Section 4. Section 5 surveys the relevant research. Finally, Section 6 provides conclusions and discussion of future work.

2. BACKGROUND

At the class testing level, a testing model has to be constructed and used to generate the test cases. In [8], a novel technique to synthesize a class state-based testing model from the specifications (i.e., preconditions and postconditions) of class methods is introduced. Al Dallal et al. [9] introduced a test case generation criterion called all paths-state that uses the synthesized model to generate test cases that are effective at the application testing stage in covering the reused FIC specifications.
2.1. Framework hooks

In [7,10], the issue of documenting the purpose of a framework and how it is intended to be used using the hooks is described and formalized. Hooks describe how to extend or customize parts of the framework to build an application.

Froehlich [7] provided a special purpose language and grammar in which the hook description can be written. Each hook description consists of the following parts. (1) A unique name. (2) The requirement (i.e., the problem the hook is intended to help solve). (3) The hook type. (4) The other hooks required to use this hook. (5) The components that participate in this hook. (6) The 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 made to develop the application. (8) The postconditions (i.e., constraints on the parameters that must be true after the hook has been used). (9) A general comment section. It is not necessary to have all the above parts for each hook.

Figure 4 shows a hook description example for the creation of an account in a banking framework. The Initialize Account hook creates a constructor method for the NewAccount class (i.e., a FIC defined in the framework hooks). In the constructor method, the account money currency is selected. There are three pre-built classes in the framework for money: USMoney, EURMoney, and Money. Moreover, the user has to specify the bank branches in the system. Finally, the user has to specify the maxPeriod variable value.

**Figure 4 about here, please**

Figure 4. Description of the Initialize Account hook of a banking framework.

The introduced hook description supports the framework application test design. The hook description identifies the FICs and their methods. In addition, it identifies the preconditions and postconditions of the FIC methods. These preconditions and postconditions are essential to determine the FIC behaviors and sequential constraints. Moreover, postconditions hold the expected outputs. The preconditions and postconditions of a method are called method specifications. When a FIC extends a framework class (i.e., in case of using a white-box framework), the inherited methods are either used in the context of the FIC without modifications or extended. For both cases,
the hook descriptions show how to use the inherited methods of the framework classes and identify their pre- and postconditions in the context of the FICs. When a FIC uses a framework class (i.e., in case of using a black-box framework), there are no methods inherited from the framework classes. In this case, the hook descriptions introduce methods for the FICs and show how to use the introduced methods. The technique proposed in [8] can be used to synthesize the class testing models for the FICs from the method specifications provided in the hooks. In addition, the method specifications can be used to evaluate the results of the test cases as proposed in [11].

2.2. Automatic construction of a class state-based testing model

Building a testing model to express the behavior of a class is an essential step for the generation of the class-based test cases. In [8], a technique to synthesize the class state-based testing model from the method specifications is introduced. State-based models describe software in terms of states and transitions. The state of an object of a class is an abstraction that models a set of instance variable value combinations that share some property of interest. Two special states have to be presented in any object state-model: alpha and omega. Alpha state represents the object before being constructed. The omega state represents the object after being destructed. A transition is an allowable two-state sequence. Each transition may be associated with (1) an event (i.e., a call for a class method), (2) a set of predicates, and (3) a set of expected actions. To execute a transition, the object must be in the accepting state of the transition, the event is executed, and the predicates evaluate to true.

The state-transition model synthesis technique first extracts the Class Under Test (CUT) states represented by state-invariants (i.e., assertions that define a state as combinations of instance variables) using the method specifications that depend on the instance variables of the class. Then, the technique extracts the transitions between the extracted states. To extract the states of the CUT, we have to construct a condition/instance-variable table. In the table, the columns and rows represent the instance-variables of the CUT and the precondition/postconditions, respectively. A set of
rules is applied to optimize the table. Each row in the resultant optimized table includes invariants of a state.

To extract the transitions of the CUT, the preconditions and postconditions of the CUT methods are mapped to the extracted state-invariants. Each state in which its state-invariants satisfy the preconditions of a method is a source state for the transition associated with the method call. Moreover, each state in which its state-invariants satisfy the postconditions of a method is a destination state for the transition associated with the method call. After that, the transition synthesis technique extracts the predicates and actions of the transitions by comparing the invariants of the source and destination states of the transitions with the specifications of the methods called in the events associated with the transitions. Table 1 and Figure 5 show the transitions and State-Transition Diagram (STD), respectively, of the NewAccount FIC example. The model consists of 21 transitions and six states including alpha and omega states.

Table 1 about here, please

Table 1. Transitions of the NewAccount FIC.

Figure 5 about here, please

Figure 5. STD of the synthesized state-transition model of the NewAccount FIC.

The introduced testing model synthesis technique does not automate the synthesis of the non-event-driven transitions (i.e., transitions not associated with events, e.g., transitions 20 and 21 shown in the last two rows of Table 1). For the non-event-driven transitions, it is required to determine the source and destination states first. The state-invariants of the destination state, which are different than the state-invariants of the source state, are then added as predicates to the transition. In addition, the introduced testing model synthesis technique does not guarantee synthesizing a free of infeasible paths model. Infeasible paths are the ones that cannot be executed. For example, in the STD of Figure 5, given that “amount>0” is true, the transition labeled as “7” causes several infeasible paths since depositing a positive amount of money cannot cause the balance that has a positive value to have a negative value. To solve this problem, we have to either detect the infeasible paths and avoid using them in generating the test drivers [12] or we have to ignore any test driver that has violated preconditions [13].
2.3. All paths-state test case generation technique

At the application developments stage, the application developer may implement part of the specification introduced by the framework hooks and decide that the rest of the specification is not required to be implemented and used in the application. This may affect the baseline test cases generated from the full specification provided through the hook descriptions. Therefore, the unaffected test cases may not be sufficient to cover all implemented transitions in the specification model of the FIC under test. In [9], the problem is solved by introducing a specification coverage criterion that produces test cases that are sufficient to cover all reused transitions in the modified specification models of the implemented FICs under test. The introduced coverage criterion is called all paths-state and it is used to construct a set of test cases $T$ from a specification graph $SG$ (e.g., UML statechart or finite state machine of the FIC under test). $T$ covers all simple paths to each state in the $SG$. A simple path includes only an iteration of a loop, if a loop exists in some sequence.

The set of paths that satisfy the criterion can be shown in a tree. The construction process of the tree starts from the alpha state of the $SG$. In the process, whenever a state is reached all outgoing transitions from the state are traversed. The process terminates when each root-leaf tree path terminates at the final (i.e., omega) state or a state already encountered on the path.

Figure 6 shows the all paths-state tree of the STD of Figure 5. In the STD, if any transition is deleted, reachable states from the deleted transition can still be reached by some other paths of the tree. For example, if all paths-state technique is used to build the test cases and the application developer chooses not to implement the transition originated from the $Open$ state and ended at the $Inactive$ state, the test cases that include the transition are considered broken and, therefore, they cannot be used as-is. This results in breaking the test cases built from the paths that include the transition sequences labeled as $(1,20,13,21)$, $(1,20,13,14)$, $(1,20,13,19)$, $(1,20,13,5)$, $(1,20,15)$, $(1,20,18)$, and $(1,20,4)$. Note that the remaining test cases still cover all outgoing transitions from the $Inactive$ state and, therefore, can be deployed.

Figure 6 about here, please
Figure 6. All paths-state tree of the STD example shown in Figure 3

Test cases are generated by traversing each path in the tree from the tree root to a leaf node. The number of generated test cases is equal to the number of leaf nodes in the tree. The number of leaf nodes in the tree shown in Figure 6 is 22 and, therefore, the number of generated test cases is 22. Figure 7 shows two Java test case examples generated from the tree shown in Figure 6. The two test cases are generated by traversing the paths that include the transition sequences labeled as (1->2) and (1->12->14), respectively.

Figure 7 about here, please

Figure 7. Two test case examples generated from the tree shown in Figure 6.

3. USING THE REUSABLE TEST CASES

When application developers use FICs to implement their applications, they deal with the specifications of the FICs introduced by the hooks in four ways: (1) use them as defined, (2) ignore specifications for the behaviors that are unnecessary in implementing the application requirements, (3) modify the specifications, and (4) add new specifications for the added behaviors to meet the application requirements. This paper addresses the first three ways of using the specifications introduced by the hooks. Adding new specifications requires either augmenting the reusable test cases or building new test cases from scratch. The first three ways of using the FIC specifications create the following four main problems that have to be solved to apply the reusable test cases effectively.

3.1. Tackling the ignored specifications problem

Application developers have the flexibility to ignore FIC specifications introduced by the hooks if these specifications are unnecessary in implementing the application requirements. The transitions that model the ignored specifications have to be removed from the FIC state-model. All paths-state coverage criterion produces test cases such that if a transition is removed and, therefore, test cases are broken, the remaining test cases
still cover the remaining used transitions. Therefore, no test cases should have to be created to test any of the reused transitions.

To find the broken test cases, it is required to associate the Ids of the test cases with the FIC model transitions that they cover at the framework development stage. The procedure given in Figure 8 shows how to associate the Ids of the test cases with the FIC transitions. Steps 1 and 2 of the procedure associate with each link in the all paths-state tree, the Ids of the test cases that cover the paths that contain the link in the tree. Steps 3 and 4 associates the Ids of the test cases with the transitions of the FIC model by mapping the links of the tree to the transitions of the FIC model.

**Figure 8 about here, please**

Figure 8. Associating the test case Ids to the transitions of the FIC model.

In our concrete example, each path in the tree shown in Figure 6 is used to build a test case as discussed in Section 2.3. Table 2 shows the test case Ids and the corresponding paths of the tree. Figure 9 shows the resulting all paths-state tree when the test case Ids are assigned to its links according to the first two steps of the procedure shown in Figure 8. For example, the link from the node labeled Frozen to the node labeled Inactive in the path (Alpha->Open->Frozen->Inactive) in the tree shown in Figure 9 is associated with the test case Ids 8, 9, 10, and 11 because the link is contained in the tree paths covered by these test cases. Figure 10 shows the resulting STD of the NewAccount FIC when the test case Ids are assigned to the transitions according to Steps 3 and 4 of the procedure shown in Figure 8. For example, the transition from Frozen to ω states is represented twice in the all paths-state tree shown in Figure 9 by links covered by the test cases that have Ids 13 and 17, respectively. Therefore, the transition is associated with a set of test case Ids {13,17}. The test case Ids associated to each transition are shown also in the fifth column of Table 3.

**Table 2 about here, please**

Table 2. Ids of the test cases built from the paths of the tree of Figure 6.

**Figure 9 about here, please**

Figure 9. All paths-state tree of the NewAccount FIC with Ids of the test cases assigned to the links.

**Figure 10 about here, please**
Figure 10. The STD of Figure 5 with Ids of the test cases assigned to the transitions.

**Table 3 about here, please**

Table 3. Test case Ids associated with the transitions of the STD of Figure 5.

At the application development stage, when the application developer ignores FIC specifications, the transitions in the testing model corresponding to the ignored specifications are removed. In addition, the transitions no longer contained in any path from Alpha to Omega states are removed. The broken test cases are the ones whose Ids are associated with the removed transitions.

Let us look at an example. Suppose the application developer decides to remove the transition that has an Id 21 in Table 3. From the fifth column of Table 3 we see that the test cases that have Ids 8, 9, 10, 11, and 15 cover the deleted transition and, therefore, they are broken. The Ids of the broken test cases have to be removed from all cells in the fifth column of Table 3. The results of these removals are shown in the last column of Table 3. The table shows that each of the remaining transitions is covered by at least one test case.

In conclusion, providing the framework with the description of the FIC model that has test case Ids associated with its transitions facilities detecting the broken test cases when the application developer ignores FIC specifications.

### 3.2. Tackling the renaming problem

One of the problems in reusing the test cases is that the test cases use the method names introduced by the hooks, while the actual implementation to be tested uses method names introduced by the application developer. For example, when the application developer of the banking system framework implements the `NewAccount` class he might rename it `MyAccount` and change some of the names of the class methods as shown in Table 4. Test cases generated at the framework development stage would not use the new class and method names and, therefore, could not be reused as is directly.

**Table 4 about here, please**

Table 4. Method-name-mapping table for `MyAccount` class.
To solve this problem, a mapping class that has the same name as the FIC class defined in the hooks has to be built. The mapping class inherits the implemented class (e.g., MyAccount class) and its methods map the methods introduced by the framework hooks to the ones used in the actual implementation of the class. The mapping is achieved by using a method-name-mapping table as illustrated in Table 4. Given the method-name-mapping table, the generation of the mapping class is straightforward and can be easily automated: whenever a method listed in the first column of the method-name-mapping table is invoked by a test driver (i.e., implementation of a test case), the invoked method in the mapping class invokes the corresponding method listed in the second column of the table. For example, Figure 11 shows the NewAccount mapping class that uses the method-name-mapping table shown in Table 4.

**Figure 11 about here, please**

Figure 11. NewAccount mapping class.

In MyAccount class, the constructor method is renamed to match the name of the new class name as shown in Table 4. Therefore, when test drivers call NewAccount constructor method, MyAccount constructor method is called. In Java, the renaming problem is not a problem for the constructor methods, because the constructor method of the superclass is always invoked using the super keyword regardless of the superclass name. However, the problem has to be solved as illustrated above when methods other than the constructor method are renamed. For example, balance method is renamed as getBalance in MyAccount class. When test drivers call balance method of the NewAccount class, the method invokes the getBalance method as shown in Figure 11.

The mapping class is also useful when the application developer does not implement an instance variable and its access method. If the access method of the instance variable is used in the reusable test drivers to check the state-invariants and is not contained in the implemented FIC, the reusable test drivers would not compile. In this case, the access method of the instance variable has to be implemented in the mapping class. The method returns any value accepted at any of the original FIC model states remained in the modified model. For example, suppose the application developer who uses the NewAccount FIC does not implement the frozen instance variable and its access method isFrozen(). This causes the Frozen state and the transitions associated to it to be removed.
from the STD shown in Figure 5. Despite the fact Test Case # 1 shown in Figure 7 does not cover any of the removed transitions, it does not compile because it uses the isFrozen() method not contained in the implemented FIC. To solve this problem, the mapping class should implement the isFrozen method as follows:

```java
public boolean isFrozen(){  return false; }
```

The method returns false because false is the accepted value of the frozen instance variable in the STD states shown in Figure 5 and remained in the model that represents the implemented FIC.

In conclusion, we can reuse the test drivers generated at the framework development stage as-is (i.e., without modifying them) by using a mapping class in which maps the methods invoked in the test drivers to the methods used in the actual implementation of the FIC. In addition, the mapping class implements access methods used in the test drivers to check the state-invariants and not contained in the implemented FIC.

### 3.3. Tackling the different implementations of a FIC method problem

In some cases, the application developer may decide to have different implementations for a method introduced by the hooks. For example, suppose that the application developer of the MyAccount class decides to have two implementations for the deposit method introduced by the banking framework hooks: one for depositing US money and the other one for depositing EUR money. These different implementations have common preconditions and postconditions introduced by the hooks because they are constructed using the same hooks.

To test the different implementations, the test drivers that test the method should be exercised as many times as the number of implemented versions of the method. To do so, a SwitchKey global variable accessed by both the mapping class and the class that invokes the test drivers is used to keep track of the order of the version to be called when the test drivers are exercised.

As an example, suppose that the application developer of the MyAccount class implements two versions of the deposit method as indicated earlier in Table 4. Each version has a different method name. However, both versions have common
preconditions and postconditions provided in the hooks for the deposit method and, therefore, the reusable test cases generated for the deposit method have to be applied for both implemented versions. Thus, the following code is included in the NewAccount class as shown in Figure 11:

```java
public void deposit(amount) {
    switch((new DRIVER_MyAccount).getSwitchkey()) {
    case 1:USdeposit(amount);
            break;
    case 2:EURdeposit(amount);
            break;
    }
}
```

In the above code, the return value of the getSwitchKey method is used to determine at run time which deposit method implementation is to be invoked. The getSwitchKey method is defined in the class driver that invokes the test drivers as will be illustrated in Section 3.5.

This way of testing the different implementations of the same FIC method allows for reusing the test drivers generated at the framework development stage as-is to test the different implementations of the FIC methods. No test drivers have to be created from scratch to solve the problem thereby reducing the application class testing time.

3.4. Tackling the method parameter update problem

Application developers have the flexibility to add or remove parameters from the parameter list of the FIC methods introduced by the hooks as long as they do not change the preconditions and postconditions introduced in the hooks. When an application developer removes one or more parameters from the implemented version of the method introduced by a hook, the unused parameters are just ignored at the time the test drivers invoke the method introduced by the hook. As an example, suppose that the MyAccount class is implemented. In the implemented version of the withdraw method, the parameter of the withdraw method introduced by the hook is removed to allow for withdrawing fixed amounts of money. In the implementation of the withdraw100 method, the
application developer decided to pass the parameter value hard-coded to the `super.withdraw` method as follows:

```java
public class MyAccount extends Account {
    ...
    public void withdraw100() {
        super.withdraw(100);
        ...
    }
    ...
}
```

In this case, as shown in Figure 11, the `NewAccount` mapping class ignores the parameter value passed to the `withdraw(float)` method of the class when the `withdraw100()` method is invoked.

When the application developer adds more parameters to the parameter list of a method introduced by a hook, the application developer has to pass a hard-coded value to the added parameters when the method is invoked in the class that inherits the implemented class. The application developer has to determine the values to be passed to such parameters. If more than one test value has to be exercised, the application developer has to find the test drivers that invoke the method and execute them with the other test values of the parameter.

As a conclusion, instead of modifying the reusable test drivers, the application developer can use the mapping class to solve the method parameter update problem, which reduces the cost of using the test drivers at the application development stage.

### 3.5. Invoking test drivers

Finally, it is required to build a class driver for each implemented FIC to invoke the non-broken reused, augmented, and new test drivers that test the FIC. If the `switchKey` global variable is required to allow for testing the different implementations of a FIC method as illustrated in Section 3.2, the variable is defined as a global variable in the class driver. In Java for example, the `switchKey` variable is declared private and static and an access method is implemented to get the variable value. For example, part of the class driver for the `MyAccount` test drivers is shown in Figure 12. As indicated in Section 3.4, the `deposit` method introduced in the hooks has two different implementations in the `MyAccount` FIC.
Therefore, as shown in Figure 12, the class driver declares the `switchKey` instance variable and its access method (i.e., `getSwitchKey()`). From the second and fifth columns of Table 3 it is found that `deposit` method is covered using the test cases that have Ids 2, 4, and 5. As depicted in Figure 12, the test cases that have Ids 2, 4, and 5 are exercised twice: once after the value of the `switchKey` variable is set to “1” and once after the value of the variable is set to “2”.

**Figure 12 about here, please**

Figure 12: Part of the DRIVER_MyAccount class.

Figure 13 shows the class diagram that represents the relation between the implemented FIC under test, the mapping class, the test drivers, and the class driver. The mapping class extends the FIC under test and the test drivers depend on the mapping class. Finally, the class driver depends on the test drivers. After implementing the class driver, the driver has to be executed to perform the actual testing.

**Figure 13 about here, please**

Figure 13. The class diagram of the FIC under test and the class required for the testing.

### 3.6. Fault detection

The reusable test cases are generated using all paths-state technique. At the application development stage, some of the reusable test cases are broken and cannot be used. Other reusable test cases are either used as-is or augmented. In some cases, new test cases are created from scratch to test new specifications. The following properties show that all paths-state coverage subsumes round-trip path coverage in terms of path coverage and compares the fault coverage of the resulting test cases applied at the application development stage to test the implemented FICs with the fault coverage of the round-trip path test cases. In [14], it is shown that the round-trip path test cases are reasonably effective in terms of fault coverage (i.e., 87% average fault coverage). FICs are problem domain classes, which are often suitable for testing with the round-trip path technique [4].

**Property 1:** In terms of path coverage, the all paths-state coverage subsumes the round-trip path coverage.
**Rationale:** The coverage of each of the all paths-state and round-trip path strategies is represented by a tree. The only difference between the construction procedures of the two types of trees is in the stopping criterion. In round-trip path strategy, each path in the tree ends in either a node that represents the omega state in the model or a node that represents a state in the model already *represented elsewhere in the tree*. In the all-paths-state strategy, each path in the tree ends by either a node that represents the omega state in the model or a node \( n \) that represents a state in the model already *represented elsewhere in the path that contains node \( n \).* As a result, the stopping criterion imposed by the all paths-state strategy is more constrained than the stopping criterion imposed by the round-trip path strategy. Consequently, each path in the round-trip path tree is identical to a sub-path in the all paths-state tree. Therefore, all paths-state coverage subsumes round-trip path coverage. Figure 14 shows the path coverage hierarchy. The all paths-state coverage criterion covers the same or more paths than the round-trip path coverage criterion. The all paths-state coverage criterion covers the same or less paths than the exhaustive all paths coverage criterion that covers all possible paths in a state machine.

**Figure 14 about here, please**

Figure 14. Path coverage hierarchy.

**Property 2:** After removing the broken test cases, augmenting some test cases, using some test cases as-is, and creating some test cases from scratch, the resulting test cases applied at the application development stage to test the implemented FICs have at least the same fault coverage as the round-trip path test cases.

**Rationale:** By definition, the all paths-state tree covers all simple transition sequences to each state in the state model. When a transition is deleted, the paths that include it are broken. Therefore, the rest of the paths in the resulting tree cover all simple transition sequences to each state in the state model except for the sequences that include the deleted transition. This means that the resulting tree covers all simple transition sequences to each state in the updated state model and, therefore, it is an all paths-state tree. Property 1 states that all paths-state coverage subsumes the round-trip path coverage in terms of path coverage. Therefore, for the used as-is transitions, the non-broken test cases covered by the resulting all paths-state tree have at least the same fault coverage as the round-trip path test cases that cover the reused transitions.
The transitions added to the state model because of the new code added by the application developer can be covered using the round-trip path strategy. As a result, after removing the broken test cases, augmenting some test cases, using some test cases as-is, and creating some test cases from scratch, the resulting test cases applied at the application development stage to test the implemented FICs have at least the same fault coverage as the round-trip path test cases.

4. EVALUATION

This section studies experimentally the proposed techniques in terms of test case specification coverage. The specification coverage is measured in terms of the number of transitions in the state model. Eleven applications developed using two frameworks are used in the case study.

In the case study, the automatic construction technique of the class-based testing model was used to build the class state-based testing models for the FICs of the frameworks using the hooks and other framework documents in case of having missed hooks. The all paths-state technique was used to generate test cases from the testing models of the FICs.

For the selected framework applications, for each implemented FIC, we have modified the constructed testing model of the FIC according to the specifications of the implemented FIC. This results in breaking some test cases. Finally, the number of transitions in the modified testing models covered by the non-broken test cases are counted and reported. The results show that, on average, a high percentage of the implemented FICs, in terms of the number of transitions in the testing models, in the randomly selected framework applications are tested using the reusable test cases generated at the framework development stage, which reduces the application class testing time considerably.

4.1. Testing Client Server Framework (CSF) applications
CSF [15] 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 (without comments/blank lines).

The CSF hooks describe the behavior of ten FICs and show how they can be implemented or customized. However, the set of available hook descriptions does not describe how to use all the methods of the extended framework classes. This forced us to read the Javadoc document of the extended framework classes and even to go through the framework code and communicate with the framework developer to write the preconditions and postconditions of the FIC methods inherited from the framework classes and not specified in the CSF hook descriptions. Specifically, the set of available hook descriptions define 66 methods out of 122 for the ten FICs. After that, we have applied manually the testing model synthesis technique explained in Section 2.2 to construct the testing models for the ten FICs. The synthesized models consist of a total of 94 states and 2261 transitions. Finally, we determined the paths required to build the test cases using the all paths-state technique. The actual test cases were not built because they are not required to derive the required results and because it takes a considerable time, in the absence of a supporting tool, to build test cases for the complex CSF FICs using the all paths-state technique.

Five CSF applications developed by fourth-year undergraduate students at the University of Alberta were randomly selected out of a pool of 15 applications. First, we have identified 189 implemented FICs in the applications. For each implemented FIC, we have extracted the modifications performed on the testing models. The modifications are extracted by analyzing the specifications of the implemented FICs if available or the code otherwise. This results in having 18,156 transitions in the testing models of the implemented FICs. After that, we have followed the approaches introduced in this paper to deal with ignored specifications, updated method names and parameters, and different
method implementations. Finally, we have counted the transitions in the testing models of the implemented FICs that are covered by the identified reusable test cases.

Table 5 shows the transition coverage results when the proposed techniques were applied for testing the implemented FICs in the CSF applications. The second column of the table provides the total number of classes implemented at the application development stage and does not include the number of used framework classes. The third column provides the number and percentage of implemented FICs in the applications. The implemented FICs are part of the classes implemented at the application development stage. The fourth column gives the total number of transitions in the state-transition model of the implemented FICs in the applications. The fifth column provides the total number and percentage of the implemented FIC transitions covered by the reusable test drivers as-is (i.e., with no augmentation).

Table 5 about here, please

Table 5. The results of using the CSF reusable test cases for testing CSF applications.

4.2. Testing SalesPoint framework applications

SalesPoint [16] is a framework written in Java and developed to create point-of-sale simulation applications such as a ticket vending machine application or a big supermarket with many departments application. The framework supports the management of the relations between the business, the customers, and the administrative tasks like accounting. The SalesPoint framework consists of 161 classes and it is provided with hooks that describe the behavior of 78 FICs.

In this case study, it was found that only twenty FICs out of the 78 FICs introduced by the framework hooks were used in the considered framework applications. The testing models of the twenty FICs consist of a total of 70 states and 1552 transitions.

Six SalesPoint framework applications developed by second year undergraduate students at the University of the Federal Armed Forces Munich were randomly selected in this case study. Table 6 shows the reusable test case coverage results when the proposed techniques were applied for testing the implemented FICs in the applications developed using the SalesPoint framework.
Table 6. The results of using the SalesPoint framework reusable test cases for testing the SalesPoint framework applications.

4.3. Comments on results

The results show that, on average, a considerable part of the implemented application classes (i.e., an average of 41.4% and 68.5% for the applications developed using the CSF and SalesPoint framework, respectively) is composed of FICs, which makes it worthwhile to build reusable test cases at the framework development stage for the applications developed using each of the CSF and SalesPoint framework. Moreover, the results show that, on average, a high percentage of the specifications of the FICs, in terms of the number of transitions in the state-transition model, (i.e., an average of 76.9% and 96% for the implemented FICs in the applications developed using the CSF and SalesPoint framework, respectively) are tested using the reusable test cases without modifying them. The coverage of the rest of the transitions requires augmenting some of the reusable test drivers. None of the transitions requires building test drivers from scratch.

On average, the percentage of the specifications of the FICs, in terms of the number of transitions in the state-transition models, covered using the reusable test cases without modifying them is greater in the SalesPoint framework applications than in the CSF applications. Given the relative numbers, we believe that this is because that the amount of functionality already defined for the FICs of the SalesPoint framework by the inherited framework classes or by the hooks is larger than the amount of functionality defined for the CSF FICs. One way to measure the amount of functionality is by calculating the number of methods per FIC. For the FICs of the SalesPoint framework considered in the case study, the total number of methods defined for the twenty FICs is 570 (i.e., an average of 28.5 methods/FIC), while the total number of methods defined for the ten FICs defined in the CSF hooks is 122 (i.e., an average of 12.2 methods/FIC). This gives an indication that, in general, as the amount of functionality of the FICs defined in the inherited framework class or by the hooks increases, the amount of functionality added
by the application developer to the FICs decreases. Consequently, as the amount of functionality of the FICs defined in the inherited framework class or by the hooks increases, the portion of the FICs tested by the reusable test cases as-is at the application development stage increases.

Although the applications used in the experiment are not large, they are representative with respect to the percentage of FIC transitions covered by the test cases as-is, and therefore, reflect the level of reusability we might typically expect for a family of applications using a common framework. Three months were spent by one researcher to manually build the FIC models, identify the reusable test cases, and find the number of FIC transitions covered by the test cases for the eleven applications considered in the case study. Since a large portion of the FIC testing process can be automated, the effort spent in the experiment could be reduced substantially with the aid of a supporting tool. In this case, more experiments need to be run using both a control group and the tool to determine the real effort reduction resulting from our approach.

5. RELATED WORK

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

In regression testing, a modified version of the software is tested to provide confidence that the changed parts are behaving as intended and the unchanged parts are not affected by the modifications in an unforeseen way. The test suite used to test the original version of the software or part of it is reused to test the modified version. In attempting to reuse the test suite or part of it, two problems have to be tackled: which test cases of the original test suite can or should be used to test the modified version and which new test cases must be developed to test parts of the modified software [17]. A number of regression testing techniques have been developed for testing object-oriented applications (e.g., [17,18,19,20,21,22,23,24]). As far as we know, all regression testing
techniques determine the reusable test cases by analyzing the source code. In this paper, the reusable test cases are determined by analyzing the specification models.

In subclass testing, 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 [25], the knowledge of how the subclass is derived from the superclass is used to determine where the superclass test suite must be changed and which superclass test cases have to be rerun to test the subclass. In [26], it is shown how the superclass test suite should be changed to test the subclasses and a framework is introduced to execute the test cases and check their results. In [27], the Test Template Framework, a framework for specification-based testing, is extended to include inheritance. Conditions under which superclass test cases can be reapplied as-is or after some modifications are identified. In [28], JUnit, a framework for Java unit testing, is extended such that superclass test cases are automatically extracted in subclass drivers. In subclassing, superclass features are inherited without modifications, redefinition, or extension. Therefore, the problem of detecting broken test cases for ignored specifications is not applicable and, therefore, not studied in reusing superclass test cases.

In testing the use of the class libraries and the frameworks, Binder [4] stated that the class library user and the framework user can reuse the class libraries test suite and the framework test suite, respectively, at cluster testing level without introducing new specific approaches.

In [29] and [30], the issue of product-line testing is considered. It is suggested to build general reusable test cases, associate them with the software specification, store them in a database at the product line level, and then specialize the test cases for each product. This way of using the test cases is called vertical reuse. In addition, each time 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. Whenever, the product components are used in building other products, the stored test cases are reused as-is. This way of using the test cases is called horizontal reuse. Due to the generality of the product-line testing problem, it is difficult to introduce specific techniques for generating and using reusable test cases for all types of software product-line.
In [31], it is suggested to put built-in-tests, in the form of methods that provide information to test the component, inside the reusable components and to build component testers. 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. Three stages in which the component testers can be used are introduced: when the component under test is deployed, during normal execution, and in maintenance. This technique works well for the components used as-is, but it does not work for the components that have their specifications modified at deployment time. In this case, the modifications of the component specifications have to be reflected in the reusable component testers. In [32], built-in tests are incorporated into object-oriented frameworks and used when the framework classes are inherited at the application development stage. The approach limits the reusability of the built-in tests to verify that the inherited framework features work correctly in the context of the application classes that inherit them. The work proposed in this paper extends the reusability of the test cases provided with the framework to test the FICs. Testing the FICs includes testing the inherited features from the framework classes in the context of the FICs and testing the new FIC functionalities introduced in the hooks for the FICs. Moreover, the paper addresses using reusable test cases for testing the FICs that do not inherit framework classes.

6. CONCLUSIONS AND FUTURE WORK

The paper focuses on showing how to use the reusable class-based test cases generated at the framework development stage for testing the implemented FICs in the framework applications. Four problems in using the test cases at the application development stage are addressed and given solutions. The first solution shows how to deal with the ignored specifications to detect and remove the broken test cases. The second solution shows how to map the methods implemented at the application development stage to the ones introduced by the hooks. The third solution shows how to test the different implementations of the same method introduced by the hooks using the reusable test
cases. The fourth solution shows how to test the implemented hook method using the reusable test cases when the application developer updates the method parameters. In addition, the paper demonstrates how to build a driver for the test drivers and shows that the resulting test cases have at least the same fault coverage as the round-trip path test cases.

The effectiveness of the proposed solutions in increasing the transition coverage of the reusable test cases is evaluated using eleven applications of two frameworks. The evaluation results show that a high percentage of the classes developed at the application development stage are FICs. Moreover, the results show that a high percentage of the specifications of the implemented FICs, in terms of the number of transitions in the state-transition model, are tested using reusable test cases built at the framework development stage without modifying them.

The practicality of the approaches introduced in this paper depends on two factors: (1) automation and (2) availability of hooks. For the automation, we have developed a prototype tool to semi-automate the test case generation and use. We have found that large portion of the test case generation and use processes can be automated which makes the approach practical. However, the introduced technique assumes the availability of well-written hook descriptions. The hook descriptions are used in constructing the class testing models.

The experiments illustrated in the paper use relatively small frameworks. Generally, as the size of the framework increases, the number of FICs increases. Consequently, the number of testing models constructed and the number of the reusable test cases generated increase. However, it is important to note that all of this is expended once at the framework development stage. At the application development stage, it is not required to deal with all the testing models and reusable test cases; only the reusable test cases generated for the FICs used in the application under test has to be considered. To reduce the storage demands and maintenance and management costs for the testing models and reusable test cases, only the testing models and reusable test cases for the FICs used in the application need to be included in the application package. Others can be discarded.

Framework hooks define the collaborations between framework classes and the FICs. In addition, they define some possible collaborations among the FICs. Therefore, in
future, we plan to use the knowledge of these collaborations to build reusable integration-based test cases at the framework development stage. When the FICs are implemented at the application development stage, the reusable test cases can be used to partially test the integration between the implemented FICs and the framework classes and among the FICs.

REFERENCES


**AUTHORS’ BIOGRAPHIES**

**Jehad Al Dallal** received his B.Sc. and M.Sc. degrees in Computer Engineering from Kuwait University in Kuwait. He received his PhD degree in Computer Science from University of Alberta in Canada. He is currently working at Kuwait University, department of Information Sciences as an Assistant Professor. His research interests include software testing and software analysis.
<table>
<thead>
<tr>
<th>Transition Id</th>
<th>Source state Id</th>
<th>Destination state Id</th>
<th>Transition event</th>
<th>Transition predicates</th>
<th>Transition actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alpha Open</td>
<td>NewAccount</td>
<td>amount&gt;=0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Open Open</td>
<td>balance</td>
<td>return balance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Overdrawn Overdrawn</td>
<td>balance</td>
<td>return balance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Inactive Inactive</td>
<td>balance</td>
<td>return balance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Frozen Frozen</td>
<td>balance</td>
<td>return balance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Open Open</td>
<td>deposit balance+amount&gt;=0</td>
<td>balance=balance+amount</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Open Overdrawn</td>
<td>deposit balance+amount&lt;0</td>
<td>balance=balance+amount</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Overdrawn Open</td>
<td>deposit balance+amount&gt;=0</td>
<td>balance=balance+amount</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Overdrawn Overdrawn</td>
<td>deposit balance+amount&lt;0</td>
<td>balance=balance+amount</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Open Open</td>
<td>withdraw balance-amount&gt;=0</td>
<td>balance=balance-amount</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Open Overdrawn</td>
<td>withdraw balance-amount&lt;0</td>
<td>balance=balance-amount</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Open Frozen</td>
<td>freeze</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Inactive Frozen</td>
<td>freeze</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Frozen Open</td>
<td>unfreeze balance&gt;=0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Inactive Open</td>
<td>activate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Open Omega</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Overdrawn Omega</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Inactive Omega</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Frozen Omega</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Open Inactive</td>
<td>getUpdate()&gt;=MaxPeriod</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>Frozen Inactive</td>
<td>getUpdate()&gt;=MaxPeriod &amp;&amp; !frozen</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Transitions of the NewAccount FIC.
<table>
<thead>
<tr>
<th>Test case Id</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>α, open, open</td>
</tr>
<tr>
<td>2</td>
<td>α, open, open</td>
</tr>
<tr>
<td>3</td>
<td>α, open, overdrawn, overdrawn</td>
</tr>
<tr>
<td>4</td>
<td>α, open, overdrawn, overdrawn</td>
</tr>
<tr>
<td>5</td>
<td>α, open, overdrawn, open</td>
</tr>
<tr>
<td>6</td>
<td>α, open, overdrawn, ω</td>
</tr>
<tr>
<td>7</td>
<td>α, open, open</td>
</tr>
<tr>
<td>8</td>
<td>α, open, frozen, inactive, inactive</td>
</tr>
<tr>
<td>9</td>
<td>α, open, frozen, inactive, open</td>
</tr>
<tr>
<td>10</td>
<td>α, open, frozen, inactive, frozen</td>
</tr>
<tr>
<td>11</td>
<td>α, open, frozen, inactive, ω</td>
</tr>
<tr>
<td>12</td>
<td>α, open, frozen, open</td>
</tr>
<tr>
<td>13</td>
<td>α, open, frozen, ω</td>
</tr>
<tr>
<td>14</td>
<td>α, open, frozen, frozen</td>
</tr>
<tr>
<td>15</td>
<td>α, open, inactive, frozen, inactive</td>
</tr>
<tr>
<td>16</td>
<td>α, open, inactive, frozen, open</td>
</tr>
<tr>
<td>17</td>
<td>α, open, inactive, frozen, ω</td>
</tr>
<tr>
<td>18</td>
<td>α, open, inactive, frozen, frozen</td>
</tr>
<tr>
<td>19</td>
<td>α, open, inactive, open</td>
</tr>
<tr>
<td>20</td>
<td>α, open, inactive, ω</td>
</tr>
<tr>
<td>21</td>
<td>α, open, inactive, inactive</td>
</tr>
<tr>
<td>22</td>
<td>α, open, ω</td>
</tr>
</tbody>
</table>

Table 2. Ids of the test cases built from the paths of the tree of Figure 6.
<table>
<thead>
<tr>
<th>Transition Id</th>
<th>Event</th>
<th>Source state</th>
<th>Destination state</th>
<th>Test cases Ids</th>
<th>Test cases Ids after removing (frozen,inactive) transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NewAccount</td>
<td>α</td>
<td>open</td>
<td>1-22</td>
<td>1-7,12-14,16-22</td>
</tr>
<tr>
<td>2</td>
<td>balance</td>
<td>open</td>
<td>open</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>balance</td>
<td>overdrawn</td>
<td>overdrawn</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>balance</td>
<td>inactive</td>
<td>inactive</td>
<td>8,21</td>
<td>21</td>
</tr>
<tr>
<td>5</td>
<td>balance</td>
<td>frozen</td>
<td>frozen</td>
<td>14,18</td>
<td>14,18</td>
</tr>
<tr>
<td>6</td>
<td>deposit</td>
<td>open</td>
<td>open</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>deposit</td>
<td>overdrawn</td>
<td>open</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>deposit</td>
<td>overdrawn</td>
<td>overdrawn</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>deposit</td>
<td>overdrawn</td>
<td>overdrawn</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>withdraw</td>
<td>open</td>
<td>overdrawn</td>
<td>3-6</td>
<td>3-6</td>
</tr>
<tr>
<td>11</td>
<td>withdraw</td>
<td>open</td>
<td>overdrawn</td>
<td>8-14</td>
<td>12-14</td>
</tr>
<tr>
<td>12</td>
<td>freeze</td>
<td>inactive</td>
<td>frozen</td>
<td>10,15-18</td>
<td>16-18</td>
</tr>
<tr>
<td>13</td>
<td>freeze</td>
<td>frozen</td>
<td>open</td>
<td>12,16</td>
<td>12,16</td>
</tr>
<tr>
<td>14</td>
<td>unfreeze</td>
<td>inactive</td>
<td>open</td>
<td>9,19</td>
<td>19</td>
</tr>
<tr>
<td>15</td>
<td>activate</td>
<td>inactive</td>
<td>open</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>open</td>
<td>ω</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>overdrawn</td>
<td>ω</td>
<td>11,20</td>
<td>20</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>inactive</td>
<td>ω</td>
<td>13,17</td>
<td>13,17</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>frozen</td>
<td>ω</td>
<td>15-21</td>
<td>16-21</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>open</td>
<td>inactive</td>
<td>8-11,15</td>
<td></td>
</tr>
</tbody>
</table>

Table 3. Test case Ids associated with the transitions of the STD of Figure 5.
<table>
<thead>
<tr>
<th>Method declaration in Banking System framework hooks</th>
<th>Method declaration in MyAccount class</th>
</tr>
</thead>
<tbody>
<tr>
<td>NewAccount(float)</td>
<td>MyAccount(float)</td>
</tr>
<tr>
<td>deposit(float)</td>
<td>USdeposit(float)</td>
</tr>
<tr>
<td></td>
<td>EURdeposit(float)</td>
</tr>
<tr>
<td>withdraw(float)</td>
<td>withdraw100()</td>
</tr>
<tr>
<td></td>
<td>withdraw(float)</td>
</tr>
<tr>
<td>balance()</td>
<td>getBalance()</td>
</tr>
<tr>
<td>freeze()</td>
<td>freeze()</td>
</tr>
<tr>
<td>unfreeze()</td>
<td>unfreeze()</td>
</tr>
<tr>
<td>activate()</td>
<td>activate()</td>
</tr>
</tbody>
</table>

Table 4. Method-name-mapping table for *MyAccount* class.
<table>
<thead>
<tr>
<th>Application name</th>
<th>Number of classes</th>
<th>Number of FICs</th>
<th>Number of transitions in FICs</th>
<th>Number of FIC transitions covered by the test cases as-is</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student management system</td>
<td>47</td>
<td>31 (66%)</td>
<td>413</td>
<td>369 (89.3%)</td>
</tr>
<tr>
<td>Chatting system</td>
<td>55</td>
<td>3 (5.5%)</td>
<td>113</td>
<td>68 (60.2%)</td>
</tr>
<tr>
<td>Course management system</td>
<td>44</td>
<td>17 (38.6%)</td>
<td>456</td>
<td>322 (70.6%)</td>
</tr>
<tr>
<td>StoneClash Strategy Game</td>
<td>106</td>
<td>56 (52.8%)</td>
<td>761</td>
<td>582 (76.5%)</td>
</tr>
<tr>
<td>Army Game</td>
<td>149</td>
<td>66 (44.3%)</td>
<td>1171</td>
<td>899 (76.8%)</td>
</tr>
<tr>
<td>Average</td>
<td>80.2</td>
<td><strong>41.4%</strong></td>
<td>583</td>
<td><strong>76.9%</strong></td>
</tr>
</tbody>
</table>

Table 5. The results of using the CSF reusable test cases for testing CSF applications.
<table>
<thead>
<tr>
<th>Application name</th>
<th>Number of classes</th>
<th>Number of FICs</th>
<th>Number of transitions in FICs</th>
<th>Number of FIC transitions covered by the test drivers as-is</th>
</tr>
</thead>
<tbody>
<tr>
<td>FastFood shop system</td>
<td>18</td>
<td>13 (72.2%)</td>
<td>1147</td>
<td>1135 (99%)</td>
</tr>
<tr>
<td>Tiler shop system</td>
<td>39</td>
<td>28 (71.8%)</td>
<td>2605</td>
<td>2434 (93.4%)</td>
</tr>
<tr>
<td>Photo-service system</td>
<td>76</td>
<td>41 (53.9%)</td>
<td>3465</td>
<td>3303 (95.3%)</td>
</tr>
<tr>
<td>Casino system</td>
<td>41</td>
<td>25 (60.1%)</td>
<td>3537</td>
<td>3337 (94.3%)</td>
</tr>
<tr>
<td>Golf club system</td>
<td>50</td>
<td>45 (90%)</td>
<td>3994</td>
<td>3910 (97.9%)</td>
</tr>
<tr>
<td>Pizza shop system</td>
<td>59</td>
<td>37 (62.7%)</td>
<td>3408</td>
<td>3264 (95.8%)</td>
</tr>
<tr>
<td>Average</td>
<td>47.2</td>
<td><strong>68.5%</strong></td>
<td>3026</td>
<td><strong>96%</strong></td>
</tr>
</tbody>
</table>

Table 6. The results of using the SalesPoint framework reusable test cases for testing the SalesPoint framework applications.
Figure 1. The paper framework.
Figure 2. Framework application classes.
Figure 3. The FIC testing process.
Figure 4. Description of the *Initialize Account* hook of a banking framework.

**Name:** Initialize Account  
**Requirement:** Initialize an account (i.e., set the currency and bank branches).  
**Type:** Template  
**Uses:** None  
**Participants:** Account(framework), NewAccount(app), Amoney(app);  
**Preconditions:** \( \text{amount} \geq 0 \);  
**Changes:**

NewAccount.NewAccount(int amount) extends Account.Account(int amount);  
Choose AM from (Money, USMoney, EURMoney);  
Create Object Amoney as AM() in MyAccount. NewAccount(int);  
Create Object branches as Branches() in NewAccount.NewAccount(int);  
Repeat as necessary {
  Acquire BranchName: string  
  NewAccount.NewAccount(int) -> branch.addBranch(BranchName);
}

Acquire maxPeriod : integer domains:0-999999;  
NewAccount.NewAccount(int) -> NewAccount.setMaxPeriod(maxPeriod);  
**Postconditions:**

Operation NewAccount. NewAccount (int);  
NewAccount.balance>=0;  
! NewAccount.frozen;  
NewAccount.getUpdate()< NewAccount.MaxPeriod  
**Comments:**
Figure 5. STD of the synthesized state-transition model of the NewAccount FIC.
Figure 6. All paths-state tree of the STD example shown in Figure 3.
Test Case # 1 (covers transition sequence 1->2)
public class TEST1_NewAccount{
    public TEST1_NewAccount(){
        float amount=1;
        NewAccount o = new NewAccount(amount);
        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-o.getLastActivityDate())<
            o.getMaxPeriod()) && !(o.isFrozen()) */
        o.balance();
        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-o.getLastActivityDate())<
            o.getMaxPeriod()) && !(o.isFrozen()) */
    }
}

Test Case # 12 (covers transition sequence 1->12->14)
public class TEST12_NewAccount{
    public TEST12_NewAccount(){
        float amount=1.0;
        NewAccount o = new NewAccount(amount);
        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-o.getLastActivityDate())<
            o.getMaxPeriod()) && !(o.isFrozen()) */
        o.freeze();
        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-o.getLastActivityDate())<
            o.getMaxPeriod()) && !(o.isFrozen()) */
        o.unfreeze();
        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-o.getLastActivityDate())<
            o.getMaxPeriod()) && !(o.isFrozen()) */
    }
}

Figure 7. Two test case examples generated from the tree shown in Figure 6.
1. Assign the empty set $s$ of the test case Ids to each link in the all paths-state tree.

2. For each path in the all paths-state tree do
   2.1. Add the Id of the test case that traverses the path to the set $s$ of each link in the path.

3. Assign an empty set $e$ of the test case Ids to each transition in the FIC testing model.

4. For each link $l$ in the all paths-state tree do
   4.1. Search for the corresponding transition $t$ in the FIC testing model.
   4.2. $e_t = e_t \cup s_l$

Figure 8. Associating the test case Ids to the transitions of the FIC model.
Figure 9. All paths-state tree of the NewAccount FIC with Ids of the test cases assigned to the links.
Figure 10. The STD of Figure 5 with IDs of the test cases assigned to the transitions.
public class NewAccount extends MyAccount {
    public NewAccount(float amount) {
        super(amount);
    }
    public float balance() {
        return getBalance();
    }
    public void deposit(float amount) {
        switch((new DRIVER_MyAccount()).getSwitchkey()) {
            case 1: USdeposite(amount); break;
            case 2: EURdeposit(amount); break;
        }
    }
    public void withdraw(float amount) {
        switch((new DRIVER_MyAccount()).getSwitchkey()) {
            case 1: super.withdraw(amount); break;
            case 2: super.withdraw100(); break;
        }
    }
    public void freeze() {
        super.freeze();
    }
    public void unfreeze() {
        super.unfreeze();
    }
    public void activate() {
        super.activate();
    }
}

Figure 11. NewAccount mapping class.
public class DRIVER_MyAccount{
    private static int switchKey=1;
    public int getSwitchKey() {
        return switchKey;
    }
    public static void main(String args[]){
        /* switchKey is already set to 1 */
        /* Invoking the test drivers that cover the first implementation of the deposit method */
        new TEST2_NewAccount();
        new TEST4_NewAccount();
        new TEST5_NewAccount();
        ... 
        switchKey=2;
        /* Invoking the test drivers that cover the second implementation of the deposit method */
        new TEST2_NewAccount();
        new TEST4_NewAccount();
        new TEST5_NewAccount();
        ...
    }
}

Figure 12: Part of the DRIVER_MyAccount class.
Figure 13. The class diagram of the FIC under test and the class required for the testing.
Figure 14. Path coverage hierarchy.

All paths

↓

All Paths-state Paths

↓

All Round-trip Paths

↓

less path coverage power