International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:4, No:3, 2010

Testing Object-Oriented Framework Applications Using $FIST_2$ Tool: A Case Study

Jehad Al Dallal

***Abstract*—**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., a family of products that shares the common features). Software testing is a time-consuming and costly ongoing activity during the application software development process. Generating reusable test cases for the framework applications during the framework development stage, and providing and using the test cases to test part of the framework application whenever the framework is used reduces the application development time and cost considerably. This paper introduces the Framework Interface State Transition Tester ($FIST_2$), a tool for automated unit testing of Java framework applications. During the framework development stage, given the formal descriptions of the framework hooks, the specifications of the methods of the framework's extensible classes, and the illegal behavior description of the Framework Interface Classes (FICs), $FIST_2$ generates unit-level test cases for the classes. At the framework application development stage, given the customized method specifications of the implemented FICs, $FIST_2$ automates the use, execution, and evaluation of the already generated test cases to test the implemented FICs. The paper illustrates the use of the $FIST_2$ tool for testing several applications that use the SalesPoint framework.

***Keywords*—**Automated testing, class testing, FICs, $FIST_2$, object-oriented framework, object-oriented testing.

I. Introduction

An application framework provides a reusable design and implementation for a family of software systems [1]. 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 hooks [2].

To build an application using a framework, application developers create two types of classes: (1) classes that use the framework classes, and (2) classes that do not. Classes that use the framework classes are called Framework Interface Classes (FICs) because they act as interfaces between the framework classes and the second type of the classes created by application developers. Fig. 1 shows the relationship between the framework classes, the hooks, the FICs, and the other application classes. FICs use the framework classes in two ways: either by subclassing them or by using them without inheritance. Hooks define how to use the framework, and therefore, they define the FICs and specify the pre-conditions and post-conditions of the FIC methods. Froehlich [3] provides a special purpose language and grammar in which the hook description can be written. The hook description includes the implementation steps and the specifications (i.e., pre-conditions and post-conditions) of the FIC methods.

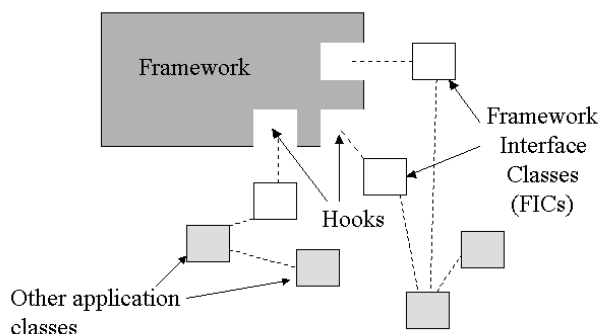

Fig. 1 Framework application classes

Software testing is a critical and important stage of the application software development life-cycle and it affects the overall software quality. In a typical programming project, approximately half of the effort is spent on testing activities [4]. However, researchers commonly limit framework reusability to only code and design. Extending the reusability to test artifacts is expected to reduce the framework application testing time and increase application quality. Building reusable test cases for the framework application during the framework testing stage increases the framework's development time and cost. However, there exists a high probability that the original investment will be recouped after producing a few framework applications. This investment cannot be fully realized unless the reusable test cases are effective and easy-to-use in testing the applications. Providing the frameworks with reusable test cases is expected to make the frameworks more marketable and provide encouragement for software developers to use them.

The Framework Interface State Transition Tester ($FIST_2$) is a tool that supports the generation of the reusable class-level test drivers (i.e., implementations of the test cases) for Java framework applications at the framework development stage and the use of the test drivers at the framework application development stage. Hooks provide the specifications of the

Manuscript received July 3, 2007. Jehad Al Dallal is with Department of Information Sciences, Kuwait University, P.O. Box 5969, Safat 13060, Kuwait (e-mail: jehad@cfw.kuniv.edu).

behaviors of the FICs. The provided specifications are provided in terms of FIC method pre-conditions and post-conditions. At the framework development stage, $FIST_2$ automatically synthesizes the state-transition testing models for the FICs and uses them to generate the reusable test drivers. These test drivers cannot be applied at the framework development stage because they are generated for classes that do not exist at the framework development stage. When application developers use FICs to implement their applications, they deal with the specifications of the FICs introduced by the hooks in three ways: (1) by using them as defined, (2) by adding new specifications for the added behaviors to meet the application requirements, and (3) by ignoring the specifications for the behaviors that are unnecessary in implementing the application requirements. Therefore, at the framework application development stage, $FIST_2$ determines broken test drivers (i.e., test drivers that cannot be run because of an ignored specification), test drivers that can be applied as-is, and test drivers that have to be augmented. Finally, the $FIST_2$ tool augments the augmentable test drivers, executes them along with the non-broken test drivers, and evaluates the actual results of the test drivers as "pass" or "no pass".

This paper introduces the $FIST_2$ tool and shows how it can be used to generate reusable test drivers for SalesPoint framework applications.

The paper is organized as follows. Section II discusses the related work. Section III introduces the $FIST_2$ tool. Section IV introduces the SalesPoint framework and illustrates the use of the $FIST_2$ tool at the framework development stages. In Section V, the use of $FIST_2$ tool at the development stage of several classes developed in applications that use SalesPoint framework. Section VI provides conclusions and a discussion of future work.

II. Related Work

In object-oriented testing, each class in the system under test has to be tested individually. Class testing is a unit testing step with respect to application testing and the first level of integration testing. At the class testing level, the method responsibilities, intraclass interactions, and superclass/subclass interactions are considered [5]. Research in generating test cases to test an implementation at the class level can be divided into two broad approaches: (1) generating test cases from the source code to achieve a given level of statement, branch, or path coverage; and (2) generating test cases from the formal specifications of the implementation. Testing techniques that follow the former approach are called implementation-based testing techniques (sometimes referred to as white box testing techniques), while testing techniques that follow the latter approach are called specification-based testing techniques (sometimes referred to as black box testing techniques).

The specification of a class behavior can be expressed using state-based models, such as finite state machines and UML statecharts [5]. State-based specifications 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. Typically, two special states have to be presented in any object state-model: alpha and omega, to represent the states of the object before construction and after destruction. A transition is an allowable two-state sequence. Each transition can 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.

There are several state-based specification coverage criteria proposed in the literature such as all-transitions [6], [7] and [8], transition-pair [6], [7] and [9], full predicate [6], [9], round-trip path [5], and all paths-state coverage criteria [10]. In software testing, it is necessary to develop oracles to evaluate the actual results of the test cases as pass or no pass. Recently, testing researchers have started to use an automatic error checking mechanism called contracts [11], [12], [13], [14], [15], and [16] as a substitute for hard-coded test oracles. Contracts are used to specify the pre-conditions and post-conditions of the class methods and the class invariants. Method pre-conditions are the conditions that must be true before the method can be executed. Method post-conditions are the conditions that must be true after the method has been executed. Class invariants are the conditions that must exist for all methods. Contracts are used at run-time to detect software defects.

There are several tools introduced to support the specification-based testing and the use of the contracts. Jcontract [14] and iContract [16] are tools used to evaluate test cases generated for Java programs using Design-by-Contract (DbC) contracts. In [15], Java Modeling Language JML [17] and [18] is integrated with the Junit framework [19] to test Java methods. JML is also used in the Korat framework [12], where method specifications are used to generate test drivers for Java methods automatically and to check the correctness of the outputs. JTest [20] is a tool that uses DbC contracts to generate test drivers for Java methods automatically and to check the correctness of the outputs. In [21], the VDM-SL specification is used to generate black box test drivers and CORBA-supported VDM oracles for CORBA-compliant programming languages. Finally, in [22], a JFramework testing environment is introduced to support testing Java frameworks using hooks. JFramework synthesizes extended FSM testing models and different implementations of the FIC methods from the hook descriptions, generates framework test drivers, and executes and evaluates them.

Several research studies focused on testing framework applications, including [23], [24], and [25]. None of proposed testing techniques is automated. In [23], it is suggested that the testing of framework applications should be based on system requirements. The new classes and objects developed by the application developer must be individually tested.

Moreover, cluster testing should be applied to verify that the developer objects are making correct use of the framework code. In this step, the framework test suite could be extended to test the application extensions. Binder neither suggests a specific methodology that makes use of the framework test suite to test the applications at the class or cluster level nor provides a discussion on which framework test suite can be extended or how a framework test suite can be extended. In [24], issues of testing applications developed with design patterns using object-oriented frameworks are discussed. It is suggested that framework developers test that the extensible patterns allow the application developer to extend its functionality. The application designers should verify that the extension points are properly coded and tested. The proposed testing techniques are limited to cluster-level testing. Finally, in [25], it is proposed to provide the framework with reusable test cases that can be applied during the application development stage. However, these test cases are limited to testing whether the inherited framework features work correctly in the context of the application classes that inherit them and do not address testing the features of the application classes.

III. The FIST$_2$ Tool

FIST$_2$ is a tool that supports the generation of the reusable test drivers for Java framework FICs during the framework development stage. It also deploys, executes, and evaluates the test drivers at the application development stage.

A. Framework Development Stage

At the framework development stage, the FIST$_2$ tool supports the generation of the reusable test drivers for Java framework FICs. The tool semi-automates the construction of the state-transition tables for the FICs, checks the correctness of the tables, and generates reusable test drivers using the all paths-state technique.

Fig. 2 shows the high-level design of the tool when used at the framework development stage. The user (typically the framework developer in a test case generation role) selects the framework. The framework is stored in a database that contains the framework code and the descriptions of the hooks. The tool passes the hook descriptions to the *FIC state-transition table-builder* module. The FIC state-transition table-builder module parses the pre-conditions and post-conditions of the FIC methods, analyzes them, and produces the state-transition table for the FIC. The framework developer can edit the generated table to add the code required to satisfy the predicates of the transitions and to add the non-event-driven transitions. The tool translates the tabular form of the state-transition model into a text and stores the text in a file in the framework database. The user can use the *Model Checker module* of the FIST$_2$ tool to check the correctness of the model in terms of connectivity and usability in building the test drivers.

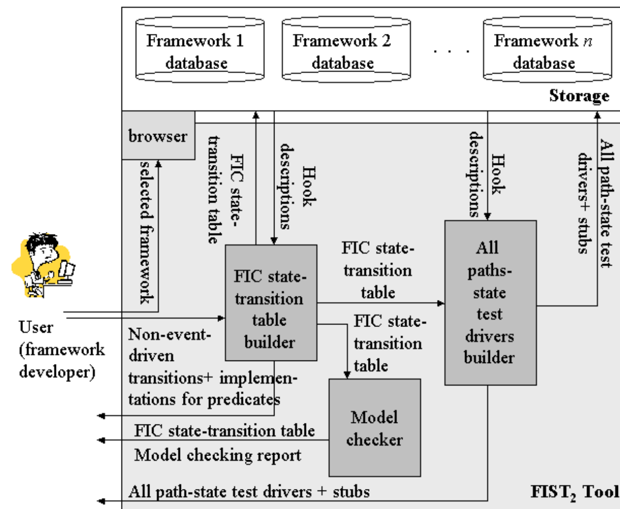

Fig. 2 The high-level design of FIST$_2$ tool (framework development stage)

The *all paths-state test drivers builder* component of the FIST$_2$ tool uses the state-transition table to generate the all paths-state test drivers and associates the test driver identifiers with the model transitions. In addition, it uses the hook descriptions to determine and generate the stubs required at the application testing stage to isolate the FICs. The test drivers and stubs are stored in the framework database and provided to the user. In the FIST$_2$ tool, the all paths-state technique [10] is used because it generates test drivers such that if some of them are broken at the application development stage because of ignored specifications, the remaining test drivers will cover the used specifications. Therefore, it eliminates the need for building test drivers from scratch to test specifications that are introduced at the framework development stage.

B. Application Development Stage

At the application development stage, the FIST$_2$ tool supports the use of the reusable test drivers generated during the framework development stage for Java framework FICs. The tool interacts with the Hook Master tool to construct the updated state-transition tables for the FICs, checks the correctness of the tables, determines the reusable test drivers, augments some reusable test drivers, and generates new test drivers to test new specifications. It then executes the test drivers and evaluates their results.

Fig. 3 shows the high-level design of the tool when used at the application development stage. The tester selects the framework stored in a database that contains the framework code, the FIC state-transition tables, and the reusable test drivers. The user uses Hook Master to semi-automate the implementation of the FICs. Hook Master comments on the Java code of the hook methods with the corresponding pre-conditions and post-conditions specified in the hook description. The pre-conditions and post-conditions are

written in the DbC language [13]. The user can add new code and specifications in DbC to the Java code to complete the implementation of the FIC. Hook Master also produces the method-name-mapping table that maps the methods defined in the hooks to the ones implemented in the FIC.

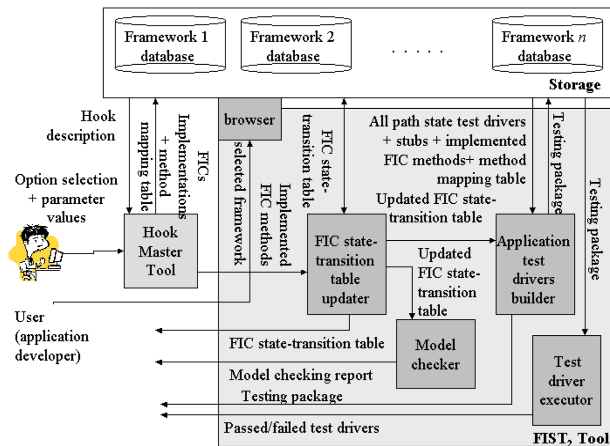

Fig. 3 The high-level design of FIST tool (application development stage)

The $FIST_2$ tool gets the used FIC methods and the new methods from Hook Master to update the FIC state-transition tables using the FIC state-transition table updater module. The user can use the Model Checker module of the $FIST_2$ tool to check the correctness of the table. The tool stores the updated table and passes it, along with the reusable test drivers, to the Application test drivers builder module, which detects broken test drivers, augments some reusable test drivers, and generates new ones to test the new specifications not covered in the augmented test drivers. In addition, the Application test drivers builder module generates a driver class for the test drivers and uses the method-name-mapping table generated by Hook Master to generate the FIC mapping class. The Application test drivers builder module also produces the necessary stubs. The generated classes and test drivers are stored in the application database.

The *Test driver executer* module of the $FIST_2$ tool compiles the test drivers and the implemented FICs using the dbc_javac compiler of the Jcontract tool [14]. The Jcontract compiler checks the DbC specifications in the Javadoc comments, generates instrumented .java files with extra code to check the contracts (i.e., pre-conditions and post-conditions) in the Javadoc comments, and compiles the instrumented .java files with the javac compiler. The resulting .class files are instrumented with extra bytecodes to check the contracts at runtime. Other classes, such as the mapping class and the driver class, are compiled using the regular Java compiler. Finally, the $FIST_2$ tool executes the test drivers and uses the Jcontract tool to check the contracts automatically at runtime and report any violations found.

IV. Generating Reusable Test Drivers for the SalesPoint Framework

SalesPoint [26] 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 (i.e., with many departments) application. The framework supports the management of relations between the business, the customers, and administrative tasks like accounting. The SalesPoint framework consists of 161 classes; it comes with hooks that describe the behavior of 78 FICs and show how they can be implemented or customized.

In this case study, it is found that only 20 FICs of the 78 FICs introduced by the framework hooks were used in the considered framework applications. The testing models of the 20 FICs consist of a total of 70 states and 1,226 transitions, including 326 transitions for illegal behavior of the FICs. In this paper, we show the use of the $FIST_2$ tool in generating the test drivers for a FIC example named NewShop, which has to be implemented in each application.

A. Generating Test Drivers for NewShop FIC

NewShop FIC is a class defined in the SalesPoint framework hooks to extend the Shop SalesPoint framework class. Shop is responsible for central management tasks and persistence. It consists of 44 public methods that operate on 21 instance variables. SalesPoint framework hooks, which define NewShop class, describe how to use 12 of the Shop class methods and do not introduce any additional methods for the NewShop class. The hooks specify the name of the FIC (i.e., NewShop), the names of the methods, the method parameters, and the method specifications. The $FIST_2$ tool parsed the hook descriptions and the Shop method specifications written in DbC and synthesized the state-transition testing model table, which consists of 5 states, including the alpha and omega states and 118 transitions. The table was shown to the user (see Fig. 4) who added the implementations required for the predicates and 41 transitions required to specify the behavior of the class due to illegal events. Then, the tool generated the test drivers according to all paths-state testing technique. The test drivers use the names of the FIC and the methods introduced by the hooks.

Fig. 5 shows a test driver example generated by the tool. Each test driver is a class that subclasses the “virtual” *NewShop* class. The test driver class contains only a constructor method in which a state-transition model path is traversed according to the all paths-state testing technique. For each traversed transition in the path, the generated code included the required implementation for the transition predicates, the call for the method associated with the transition, and the DbC Javadoc comments used at run time to check that the actions were performed correctly and the reached state (in terms of instance variable values) was as expected. Finally, the tool associated with each transition in the model the IDs of the test driver classes that traverse the transition and stored the table and the test drivers in the SalesPoint framework database.

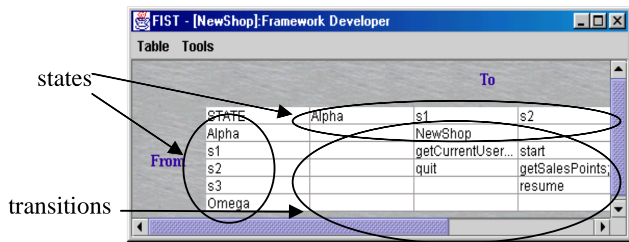

Fig. 4 Part of the state-transition table generated for the *NewShop* FIC

```
public class TEST6_NewShop{
   public TEST6_NewShop(){
      /* Test transition: source state: Alpha, sink state: s1, event:
         NewShop()*/
      NewShop o = new NewShop();
      /** @assert(o.getShopState()==o.DEAD) */

      /* Test transition: source state: s1, sink state: s2, event:
         start()*/
      o.start();
      /** @assert(o.getShopState()==o.RUNNING) */

      /* Test transition: source state: s2, sink state: s3, event:
         suspend()*/
      o.suspend();
      /** @assert(o.getShopState()==o.SUSPENDED) */

      /* Test transition: source state: s3, sink state: s2, event:
         resume()*/
      o.resume();
      /** @assert(o.getShopState()==o.RUNNING) */
   }
}
```

Fig. 5 Sample test driver generated for the *NewShop* FIC

V. Testing FICs in SalesPoint Framework Applications Using FIST$_2$

Several SalesPoint application classes developed by second-year undergraduate students were randomly selected in this case study. At the application development stage, there were several problems to be solved. This section illustrates the problems and their proposed solutions. In addition, examples are used to show how the FIST$_2$ tool is used to test selected application classes.

A. Implementing FICs

Considered applications were not developed using the introduced tool. However, we show, in the following example, a simulation development process using the tool. In the FastFood System application, when the Hook Master tool was used to create a NewShop class, the class was named FastFood. The developer interacted with the tool to override two introduced methods and to extend the constructor method. In this example, except for the constructor method, the user did not change the names of the methods introduced by the hooks. Since the FastFood class extends the Shop class, the FastFood class inherits all none overridden Shop class methods. Table I shows the mapping between the names of the implemented methods in the FastFood class and the names of the methods introduced by the hooks. The contents of Table 1 are stored to be used by the FIST$_2$ tool. When Hook Master generated part of the FastFood class code using the implementation steps provided in the SalesPoint framework hooks, it instrumented the method specifications provided by the hooks into the generated code as DbC Javadoc comments. Finally, the application developer customized some of the FastFood class methods generated by the Hook Master tool, added some DbC post-conditions for the customized methods, added one method to the FastFood class, and added the DbC pre-conditions and post-conditions for the added method.

TABLE I
Method Name Mapping Table for *FastFood* Class

Method declaration in SalesPoint framework hooks	Method declaration in the FastFood class
NewShop()	FastFood()
createShopMenuSheet()	createShopMenuSheet()
quit()	quit()

B. Using Test Drivers

After implementing the FIC, the FIST$_2$ tool uses some or all the reusable FIC test drivers provided with the framework. To make the test drivers ready for use, the FIST$_2$ tool tackles several problems. The following examples discuss the problems and explain the solution techniques.

1) Tackling the ignored specification 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. The all paths-state coverage technique 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 use the FIST$_2$ tool for testing the FastFood class, the contents of the method name mapping table, the implemented FastFood class, and the state-transition table created at the framework development stage for the NewShop "virtual" class were provided. FIST$_2$ used the contents of Table 1 to determine the reused transitions. All ignored transitions have to be removed. In addition, all unreachable states from alpha state have to be removed along with the transitions linked to them. This results in 5 states and 159 transitions in the state-transition model of the FastFood class. In this example as well as all implemented FICs in all the selected applications in the case study, none of the introduced transitions for the NewShop class were ignored. If some transitions were ignored, all test driver IDs associated with the removed

transitions would have to be removed from the list of test driver IDs associated with the remaining transitions. The set of test driver ID lists associated with the remaining transitions is the set of the reusable test drivers. All other test drivers cannot be used in testing the implemented class. This shows how the problem of finding and removing broken test drivers is solved.

2) Tackling the method renaming problem

One of the problems in reusing the test drivers is that the test drivers use the method names shown in the first column of the method name mapping table, while the actual implementation to be tested uses the method names shown in the second column of the table. To solve this problem, the $FIST_2$ tool generates a Java class that has the same name as the FIC class defined in the hooks. In the Pizza Shop system, one of the SalesPoint framework applications, the application developer implemented a FIC called NewCatalogItem and named the implemented class Order. For the Order class example, the $FIST_2$ tool generated the NewCatalogItem class, shown in Fig. 6. This class inherits the implemented class (i.e., Order class) and maps the methods introduced by the SalesPoint framework hooks to the ones used in the actual implementation of the class using the contents of the method name mapping table. For example, when the application developer implemented the Order class, the constructor method was renamed to match the name of the new class name. Therefore, when test drivers call up the NewCatalogItem constructor method, the Order constructor method should be called up as well. The renaming problem seems not to be a problem for constructor methods, because in Java, for example, the constructor method of the superclass can be always invoked using the keyword super regardless of the superclass name. However, the problem has to be solved as illustrated above when methods other than the constructor method are renamed. No example for the latter case was found in any of the SalesPoint framework applications because SalesPoint framework hooks do not introduce any new methods for the FICs.

```
public class NewCatalogItem extends Order {
    public NewCatalogItem(String st) {
        switch (SwitchKey.getSwitchKey().getSwitchkey()) {
            case 1:super(st, new Customer());
                 break;
            case 2:super();
                  break;
                       }
    }
    public CatalogItemImpl getShallowClone() {
        return super.getShallowClone();
    }
}
```

Fig. 6 *NewCatalogItem* class generated by $FIST_2$ tool

3) Tackling the different implementations of a FIC method problem

In some cases, the application developer can decide to have different implementations for a method introduced by the hooks. These different implementations have common pre-conditions and post-conditions introduced by the hooks because they are constructed using the same hooks. The different implementations can have the same method name but different parameters, or they can have the same parameters but different method names.

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, $FIST_2$ builds a SwitchKey class to keep track of the order of the version to be called when the test driver is exercised. The code of the SwitchKey class is shown in Fig. 7. For example, the application developer of the Pizza Shop system implemented two versions of the constructor method of the NewCatalogItem class. In the first version, one parameter was added to the constructor method introduced by the hook, while in the other version, the constructor method parameter was removed. Therefore, the following code is included in the NewCatalogItem class, as shown in Fig. 6:

```
public NewCatalogItem(String st)
{
   switch (SwitchKey.getSwitchKey().getSwitchkey())
   {
      case 1:super(st, new Customer());  break;
      case 2:super();  break;
   }
}
```

```
public class SwitchKey {
    private static SwitchKey SwKey;
    private int switchKey;
    public SwitchKey() {
        switchKey=1;
        SwKey=this;
    }
    public static SwitchKey getSwitchKey() {
        return SwKey;
    }
    public void setSwitchkey(int sk) {
        switchKey=sk;
    }
    public int getSwitchkey() {
        return switchKey;
    }
}
```

Fig. 7 *SwitchKey* class generated by $FIST_2$ tool

In this case, as shown in Fig. 8, before executing the test driver, the driver of the test drivers has to set the SwitchKey to decide which constructor method is to be called.

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 pre-conditions and post-conditions 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. For example, when Order class was implemented, in one version of the implemented constructor method, the parameter of the implemented constructor was removed. In the implementation of the constructor method, the application developer decided to pass the parameter value hard-coded when super method was called as follows:

```
public class Order extends CatalogItemImpl {
        public Order() {
         super("0000");
         ...          }
   ...}
```

In this case, as shown in Fig. 6, the NewCatalogItem class generated by the $FIST_2$ tool just ignored the parameter value passed to the constructor method of the class when the Order() method was invoked using the keyword super.

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. For example, when the Order class was implemented, in one version of the implemented constructor method, one parameter was added. Therefore, when the constructor method that has the additional parameter is invoked, a value was passed to the additional parameter, as shown in Fig. 6. The application developer has to determine the values to be passed on 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.

5) Tackling the test driver augmentation problem

Application developers have the flexibility to add new methods to the implemented FICs. These methods are not tested by the reusable test drivers, and therefore, they have to be tested using augmented test drivers or new test drivers created from scratch. We have identified two different effects of the added methods on the state-transition model of the FIC generated by $FIST_2$ at the framework development stage. The first effect is adding a transition to the model between two existing states. The other effect is adding states and transitions between them or between the existing states and the new ones.

The first effect requires simple augmentation. $FIST_2$ generates a round-trip path tree [5] for the new model, associates the unbroken test driver IDs linked to the transitions to the corresponding tree links, and finds the uncovered tree links. If an uncovered transition is directly linked to the alpha state, the tool generates test drivers from scratch for all round-trip paths that pass through the uncovered transition and marks all used transitions in the round-trip paths as covered. On the other hand, if the uncovered transition is not directly linked to the alpha state (i.e., there are some transitions covered in the reusable test drivers in the paths between the alpha state and the source state of the uncovered transition), the tool augments a test driver so that its ID is associated with a covered transition that has the same sink state as the source state of the uncovered transition. The tool repeats this covering algorithm until all new transitions are covered in the test drivers.

The augmentation of the test drivers to cover a transition added between existing states in the table generated at the framework development stage is called simple augmentation. This is mostly because the test drivers are augmented by just adding a call to the new method at some point in the test driver code. On the other hand, the test driver augmentation pre-formed to cover the transitions between added states and existing states is called complex augmentation. This is because considerable lines of codes have to be added to the test drivers to cover such transitions.

The developer of the Pizza Shop system added 11 methods to the Order class (i.e., the implemented version of the NewCatalogItem FIC). The $FIST_2$ tool used the DbC specifications of the added methods to update the class testing methods and used the updated model to generate the test drivers to test the added methods. In this example, all the methods were covered by the simple augmentation of some of the reusable test drivers.

6) Invoking test drivers

Finally, the $FIST_2$ tool generates a driver class for the test drivers. The driver invokes the constructor methods of the non-broken reused as-is, augmented, and new test drivers that test the implemented FIC. Part of the driver class for the Order test drivers is shown in Fig. 8. If the class that inherits the implemented class uses the SwitchKey class, the driver class of the test drivers creates an instance of the SwitchKey class and sets the key value whenever needed. In our example, the NewCatalogItem class uses the SwitchKey class, as shown in Fig. 6. Therefore, as shown in Fig. 8, an instance of the SwitchKey class is created. The tool searches for the test drivers associated with the transitions that invoke methods implemented in different versions and increments the SwitchKey value each time before invoking them. For the Order class example, the tool found that all the test drivers invoked the constructor methods. Since there are two versions of the constructor method in the implemented FIC (i.e., Order class), all the test drivers have to be invoked twice: one after setting the switchKey to "1" and one after setting it to "2". Initially, the switchKey is set to "1", as shown in Fig. 7, and therefore, no explicit statement is required to set it to "1".

After determining the reusable test drivers and augmenting some of them, the $FIST_2$ tool compiled the Order class and the test drivers using the Jcontract compiler, which translated the DbC Javadoc comments into bytecode to check the contracts.

Other classes, such as NewCatalogItem, SwitchKey, and the driver class, were compiled using a typical Java compiler. Finally, the tool executed the driver class and the Jcontract tool checked the contracts at runtime and reported the testing results.

```
public class DRIVER_ORDER{
    public static void main(String args[]){
        SwitchKey k=new SwitchKey();
        new TEST1_NewCatalogItem();
        new TEST2_NewCatalogItem();
        …
        k.setSwitchkey(2);
        new TEST1_NewCatalogItem();
        new TEST2_NewCatalogItem();
        …
    }
}
```

Fig. 8 Part of the DRIVER_ORDER class generated by $FIST_2$ tool

VI. Conclusions and Future Work

In this paper, we introduced a tool to support testing framework applications. The tool generates reusable test drivers for the FICs at the framework development stage. Moreover, the tool effectively uses the reusable test drivers at the framework application development stage to test the implemented FICs. The tool uses the method specifications provided in the hook descriptions or provided by the framework or application developers to build the testing models for the FICs and to check the correctness of the FICs at runtime.

The tool was used to generate reusable test drivers at the development stage of the SalesPoint framework. It was also used to reuse the test drivers for testing the implemented FICs in several SalesPoint framework applications.

In future, we plan to study the relationship between the size of the portion of the framework applications tested using the reusable test cases and the intersection area between the domains of the framework and the applications. Our preliminary results showed that the size increases as the intersection area between the domains of the framework and the application increases and vice versa.

References

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

[2] G. Froehlich, H.J. Hoover, L. Liu, and P.G. Sorenson, May 1997. *Hooking into Object-Oriented Application Frameworks*, Proc. 19th Int'l Conf. on Software Engineering, Boston, 491-501.

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

[4] K. Saleh, A. Boujarwah and J. Al-Dallal, Jan 2002, *"Anomaly detection in concurrent Java programs using dynamic data flow analysis"*, Journal of Information and Software Technology, Vol. 44, no 1, pp. 53-61.

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

[6] T. Chow, 1978, *Testing software design modeled by finite state machines*, IEEE Transactions on Software Engineering, EE-4(3), 178-187.

[7] J. Offutt and A. Abdurazik, October 1999, *Generating tests from UML specifications*, Second International Conference on the Unified Modeling Language (UML99), Fort Collins, CO, 416-429.

[8] K. Bogdanov and M. Holcombe, 2001, *Statechart testing method for aircraft control systems, Software Testing, Verification and Reliability*, 11(1), 39-54.

[9] A. Abdurazik, P. Ammann, W. Ding, and J. Offutt, September 2000, *Evaluation of three specification-based testing criteria*, Sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '00), Tokyo, Japan, 179-187.

[10] J. Al Dallal, 2002, *Class-based testing of object-oriented framework interface classes,* Ph.D. Thesis, University of Alberta, Department of Computing Science.

[11] L. Briand, Y. Labiche, and H. Sun, July 2002, *Investigating the use of analysis contracts to support fault isolation in object-oriented code*, International Symposium on Software Testing and Analysis ISSTA, Rome, Italy.

[12] C. Boyapati, S. Khurshid, and D. Marinov, Korat, July 2002: *Automated Testing Based on Java Predicates, International Symposium on Software Testing and Analysis* ISSTA, Rome, Italy.

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

[14] Jcontract, July 2006, http://www.parasoft.com/jsp/products/home.jsp?product= Jcontract, ParaSoft Corporation.

[15] Y. Cheon and G. Leavens, June 2002, *A simple and practical approach to unit testing: the JML and JUnit way*, Proc. of the 16th European Conference on Object-Oriented Programming (ECOOP2002), pp. 231-254.

[16] iContract: the Java Design-by-Contract tool, July 2006, http://www.javaworld.com/javaworld/jw-02-2001/jw-0216-cooltools.html.

[17] G. Leavens, A. Baker, and C. Ruby, 1999, *JML: a notation for detailed design*. In H. Kilov, B. Rupe, and I. Simmonds, editors, behavioral specifications of Businesses and Systems, chapter 12, Kluwer, pp. 175-188.

[18] G. Leavens, A. Baker, and C. Ruby, August 2001, *Preliminary design of JML: a behavioral interface specification language for Java*, TR 98-06p, Iowa State University, Department of Computer Science.

[19] Junit, July 2006, http://junit.sourceforge.net/.

[20] Jtest, July 2006, http://www.parasoft.com/jsp/products/home.jsp?product=Jtest, ParaSoft Corporation.

[21] P. Fenkam, H. Gall and M. Jazayeri, September 2002, *Constructing corba-supported oracles for testing: a case study*, Proc. of the 17th IEEE International Conference on Automated Software Applications (ASE'02), Edinburgh, UK, pp. 129-138.

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

[23] R. Binder, August 1996. *Testing for reuse: libraries and frameworks*, Object Magazine, 77-80.

[24] W. Tsai, Y. Tu, W. Shao, and E. Ebner, October, 1999. *Testing extensible design patterns in object-oriented frameworks through scenario templates*, 23rd Annual International Computer Software and Applications Conference, Phoenix, Arizona, pp. 166-171.

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

[26] The SalesPoint framework v2.0 homepage, July 2006, http://www-st.inf.tu-dresden.de/SalesPoint/v3.0/.

Jehad Al Dallal received his B.Sc. and M.Sc. in degrees in Computer Engineering from Kuwait University in Kuwait in 1995 and 1997, respectively. He received his PhD degree in Computer Science from University of Alberta in Canada in 2003.

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