Generating State-Based Testing Models for Object-Oriented Framework Interface Classes

Jehad Al Dallal, and Paul Sorenson

Abstract—An application framework provides a reusable design and implementation for a family of software systems. Application developers extend the framework to build their particular applications using hooks. Hooks are the places identified to show how to use and customize the framework. Hooks define the Framework Interface Classes (FICs) and the specifications of their methods. As part of the development life cycle, it is required to test the implementations of the FICs. Building a testing model to express the behavior of a class is an essential step for the generation of the class-based test cases. The testing model has to be consistent with the specifications provided for the hooks. State-based models consisting of states and transitions are testing models well suited to object-oriented software. Typically, hand-construction of a state-based model of a class behavior is expensive, error-prone, and may result in constructing an inconsistent model with the specifications of the class methods, which misleads verification results. In this paper, a technique is introduced to automatically synthesize a state-based testing model for FICs using the specifications provided for the hooks. A tool that supports the proposed technique is introduced.

Keywords—Framework interface classes, hooks, state-based testing, testing model.

I. INTRODUCTION

An application framework provides a reusable design and implementation for a family of software systems [1]. 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 the application developers. Building reusable test cases for the FICs and providing the test cases with the framework 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.

As shown in Fig. 1, the input to the testing process of the FICs is the specifications of the FIC methods. Method descriptions provide the specifications of the FIC methods in terms of pre- and post-conditions. There are two types of pre- and post-conditions: (1) construction ones and (2) execution ones and these are illustrated in the example in Fig. 2. The construction pre- and post-conditions are the constraints that must be satisfied before and after the hook is used, respectively, and they are identified in the hook description by keywords such as Object, Class, and Operation. The execution pre- and post-conditions are the dynamic constraints that must be satisfied before and after the methods defined in the hook are executed, respectively. The construction pre- and post-conditions, in contrast with the execution ones, do not describe the behavior of the methods defined in the hooks and, therefore, cannot be used in synthesizing the behavioral model of the FIC. The execution pre-conditions are described in terms of class instance variables and method input parameters. The execution post-conditions are described in terms of class instance variables, input parameters, output parameters, method return values, and method thrown exceptions. More precisely, the execution method specifications (i.e., pre- and post-conditions) check: (1) whether class instance variables, method input parameters, method output parameters, and method return values are within the allowed domain of values, and (2) whether the relationships among the values of the class instance variables, method input parameters, method output parameters, and method return values are satisfied.
FIC methods, which is the constructor method `NewAccount(int amount)`. In the `Pre-conditions` and `Post-conditions` sections of the hook description, the pre-conditions and post-conditions of the constructor method are specified. The first stated post-condition is a construction post-condition because it describes a condition that must be satisfied when the hook is used and it is identified by the keyword `Operation`. The other pre- and post-conditions are execution ones because they describe the conditions that must be satisfied, respectively, before and after the constructor method is executed. To determine the behavior of a FIC we have to consider all the framework hooks that specify the FIC methods. For the rest of this paper, unless stated otherwise, all pre- and post-conditions referred to are of type execution, because these are the conditions in which we are most interested in building test cases.

```
Name: Initialize Account
Requirement: Initialize an account (i.e., set the currency and bank branches).
... 
Pre-conditions: amount>=0;
Changes:
   NewAccount.NewAccount(int amount) extends Account.Account(int amount);
... 
Post-conditions:
   1. Operation NewAccount. NewAccount (int);
   2. NewAccount.balance>=0;
   3. ! NewAccount.frozen;
```

Fig. 2 The description of the `Initialize Account` hook of a banking framework

In our concrete example, the hooks of the banking framework define several public methods for the `NewAccount` FIC. The methods are `NewAccount`, `balance`, `deposit`, `withdraw`, `freeze`, `unfreeze`, `activate`. The pre- and post-conditions of the `NewAccount` FIC method defined in the hooks are listed in Table I. From the pre-conditions and post-conditions, we synthesize the states and transitions of the `NewAccount` class.

For example, in Table I, `balance()` is a method that has no pre-conditions which means that it can be called at any time during the object life cycle. `MaxPeriod` is a static instance variable, `amount` is a local variable, and the variables `balance`, `frozen`, the return of the `balance()` method, and the return of the `getUpdate()` method are non-static instance variables. The `balance()` method returns the value of the `balance` instance variable. The `getUpdate` method calculates the difference between the current date and the last activity date.

Despite the fact that the method specifications hold the specifications for the class behaviors, researchers seem to limit the method specification use to support the automated detection of software failures and the isolation of faults, and to generate method-based test cases. We are not aware of any work that uses the method specifications to generate class-based test cases. For example, in [3], to test a class behavior, class behavior testing models (e.g., state-transition model or UML statechart) used to generate the test drivers have to be pre-provided, and the method specifications are used only as testing oracles. Hand-construction of the class behavior testing model is expensive, error-prone, and may result in constructing an inconsistent model with the specifications of the class methods, which misleads verification results.

In this paper, a new technique is introduced to automatically synthesize the state-transition testing model of the FIC sequential class behavior from the specifications of the class methods. This reduces considerably the class testing cost and the chance of errors. The result is a state-based testing model that is consistent with respect to the specifications of the class methods. Therefore, using the introduced state-transition model synthesis technique, only the specifications of the FIC methods have to be provided to test the FIC behavior. The state-transition model is synthesized automatically from the method specifications. After that, a specification-based testing technique can be applied to derive the test drivers (i.e., implementations of the test cases) from the synthesized state-transition model. Finally, the test drivers are executed and the method specifications are used as testing oracles to evaluate the actual results of the test cases as pass or no pass. Fig. 3 compares the testing process that uses our proposed modeling technique and the one that does not (e.g., [4], [5], and [6]).

![](image)

**Fig. 3 FIC behavior testing process using method specifications:** (a) without using the proposed modeling technique and (b) with using the proposed modeling technique

The paper is organized as follows. Section II, discusses the related work. In Section III and Section IV, the proposed techniques to extract the class behavior states and transitions, respectively, from the method specifications are described. Finally, Section V provides conclusions and a discussion of future work.

---

**Fig. 3 FIC behavior testing process using method specifications:** (a) without using the proposed modeling technique and (b) with using the proposed modeling technique

The paper is organized as follows. Section II, discusses the related work. In Section III and Section IV, the proposed techniques to extract the class behavior states and transitions, respectively, from the method specifications are described. Finally, Section V provides conclusions and a discussion of future work.
II. RELATED WORK

Researchers identified two benefits of using the method specifications (i.e., pre- and post-conditions) (1) evaluating the results of the test cases, and (2) building test cases to test the methods of classes. The method specifications and the class invariants are called contracts. In [7], contracts are used to evaluate the results of the test cases. It is shown that contracts detect large percentage of failures (roughly 80% of the faults detected using hard-coded oracles). Moreover, it is shown that the percentage of the detected faults depends on the precision of the contracts. Baudry et al. [8] showed that the quality of the contracts is more important than their quantity. In [7], it is found that the effort involved in isolating a fault improves eight folds between programs without contracts as compared to the ones with contracts.

There are several tools and languages introduced to support the use of the contracts. Design-by-Contract (DbC) [9] is used to specify the contracts in Eiffel language. Jcontract [10] and iContract [11] are tools used to implement DbC for Java programs. In both tools, contracts written in DbC are specified as Java comments. The iContract tool instruments the Java program with extra code to check the contracts. The instrumented Java program can then be compiled as usual with a Java compiler. The Jcontract tool is provided with a compiler that checks the DbC specifications and instruments the .class file with extra bytecodes to check the contracts at run time.

In [3], Java Modeling Language (JML) [12] and [13] is used to specify the contracts for Java methods. In this work, JML is integrated with Junit framework [14] to test the Java methods. JML is also used in the Korat framework [15], where the method specifications are used to automatically generate test cases for Java methods and to check the correctness of the outputs. JTest [16] is a tool that uses DbC contracts to automatically generate test cases for Java methods and to check the correctness of the outputs.

III. SYNTHESIZING THE STATES OF A FIC

To synthesize the states of a FIC, we have to construct the condition/instance-variable table. In the table, the columns and rows represent the non-static instance-variables of the FIC and the pre-condition/post-conditions of the FIC methods that contain conditions involving the non-static instance variables, respectively. In the table, we consider only the non-static instance variables. The values of the static instance variables do not change during the object life cycle and, therefore, they do not contribute in determining the object states. Table II shows the condition/instance-variable table extracted from Table I.

Finally, the table has to be optimized by eliminating redundant rows and unnecessary information. To optimize the table follow these steps

**Step 1:** Delete any clause that depends on a dynamic variable (i.e., a variable that its value is not assigned at compilation time or can change during the object life-cycle) because the combinations of the instance variable values that represent a state of a class are determined at compilation time and do not change during the object life-cycle. The deleted clauses are considered later in the transition synthesis process.

**Step 2:** Delete redundant rows.

**Step 3:** If the combinations of instance variable values in a row \( r_1 \) overlap with the combinations of instance variable values of another row \( r_2 \), replace \( r_1 \) and \( r_2 \) with three rows: the first one contains the combinations of instance variable values contained in \( r_1 \) and not contained in \( r_2 \), the second one contains the combinations of instance variable values contained in \( r_2 \) and not contained in \( r_1 \), and the third row contains the overlapping combinations of instance variable values.

**Step 4:** Iterate through steps 2 and 3 until no redundant rows and no rows for which Step 3 can be applied exist.
Each row in the optimized table represents a state of the object of that class during its life-cycle based on the instance variable value combination shown in the row. Step 1 ensures that the states do not change during the object life-cycle (i.e., a basic property of the states of a class state-based model as described in [4]) and, therefore, each remaining clause in the table has boundary values determined at compilation time. The second step ensures that there are no redundant states in the synthesized model. Step 3 ensures that each synthesized state is exclusive (i.e., there is no overlapping states). Determining the overlapping combinations of instance variable values contained in two rows is straightforward because the condition clauses in the rows have fixed boundary values as ensured in Step 1. Finally, Step 4 holds the stopping criterion for the state synthesis process.

When the optimization rules are applied on Table II, according to Step 1, ‘balance+amount’ and ‘balance-amount’ are deleted from rows 3 and 4, respectively, because they depend on dynamic variables. This makes rows 3 and 4 redundant with row 2 and, therefore, rows 3 and 4 are deleted according to Step 2. The combinations of instance variable values contained in row 1 overlap with the combinations of instance variable values contained in row 2. Therefore, to avoid the creation of overlapping states, rows 1 and 2 are substituted with three rows. The first row contains the combinations of instance variable values contained in row 1 and not contained in row 2. This row is ignored because it does not contain any combination of instance variable values (i.e., all the combinations of instance variable values contained in row 1 overlap with the combinations of instance variable values contained in row 2). The second row contains the combinations of instance variable values contained in row 2 and not contained in row 1 (i.e., the combinations of instance variable values that has balance<0). The third row contains the overlapping combinations of instance variable values of rows 1 and 2 (i.e., the combinations of instance variable values that has balance >=0). Finally, the combinations of instance variable values contained in the first row are replaced with three rows. The first row is ignored because it does not contain any combination of instance variable values (i.e., all of the combinations of instance variable values contained in the third row formed in the previous step overlap with the combinations of instance variable values contained in row 5).

The second row has getUpdate() >=MaxPeriod and the third row has getUpdate() <MaxPeriod. This results in having the optimized table shown in Table III. In this table, each row represents a state that corresponds to the instance variable value combinations given in the row. The combinations of instance variable values in each row are called the state-invariants [4].

### IV. SYNTHESIZING THE TRANSITIONS OF A FIC

To extract the transitions that model the legal behavior of a FIC, we have to map the pre-conditions and post-conditions of the FIC methods to the extracted state-invariants. Each state in which its state-invariants satisfy the pre-conditions 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 post-conditions of a method is a destination state for the transition associated with the method call. The procedure shown in Fig. 4 explains the mapping process and shows how to extract the predicates and actions of the transitions. The source state of the constructor method is by default the alpha state. If no destructor method is specified in the class, an unlabeled transition has to be added from each state, other than the alpha state, to the omega state.
When the procedure shown in Fig. 4 is applied on tables I and III, the transitions shown in Table IV are extracted. For example, since the balance method has no pre-conditions and its post-condition satisfies all the invariants of all states, a self-loop transition associated with balance() event is added to each state (other than alpha and omega). The post-condition of the balance method is not specified in any state and, therefore, it is added as an action to all the self-loop transitions. For the withdraw method, the pre-conditions satisfy the invariants of state 1 and the post-conditions satisfy the invariants of states 1 and 2. Therefore, two transitions associated with the withdraw event are added as shown in Table IV. Note that the set of pre-conditions of the method is the same as the invariants of state 1, which causes no predicates to be added at this step. The set of post-conditions of the withdraw method includes “balance=balance-amount” which is not included in the state invariants of states 1 and 2. Therefore, the post-condition is added to both transitions as actions. Finally, since state 1 now has two outgoing transitions that have the same labels, the difference between the invariants of the destination states of the transitions has to be added as predicates to the transitions. The difference between states 1 and 2 is that state 1 has balance>=0, while state 2 has balance<0. Therefore, “balance (at destination state) >=0” has to be added to one of the transitions (from state 1 to state 1) and “balance (at destination state) <0” has to be added to the other transition (from state 1 to state 2). Since the predicates are checked at the source states, we can substitute “balance (at destination state)” by “balance-amount”.

### Table IV

<table>
<thead>
<tr>
<th>Transition number</th>
<th>Source state identifier</th>
<th>Destination state identifier</th>
<th>Transition event</th>
<th>Transition predicates</th>
<th>Transition actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>alpha</td>
<td>1</td>
<td>NewAccount</td>
<td>amount&gt;=0</td>
<td>return balance</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>balance</td>
<td></td>
<td>return balance</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>balance</td>
<td></td>
<td>return balance</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>3</td>
<td>balance</td>
<td></td>
<td>return balance</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>4</td>
<td>balance</td>
<td></td>
<td>return balance</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>deposit</td>
<td>balance+amount&gt;=0</td>
<td>balance=balance+amount</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>2</td>
<td>deposit</td>
<td>balance+amount&lt;0</td>
<td>balance=balance+amount</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>1</td>
<td>deposit</td>
<td>balance+amount&gt;=0</td>
<td>balance=balance+amount</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>2</td>
<td>deposit</td>
<td>balance+amount&lt;0</td>
<td>balance=balance+amount</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>withdraw</td>
<td>balance-amount&gt;=0</td>
<td>balance=balance-amount</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2</td>
<td>withdraw</td>
<td>balance-amount&lt;0</td>
<td>balance=balance-amount</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>4</td>
<td>freeze</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>4</td>
<td>freeze</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>1</td>
<td>unfreeze</td>
<td>balance&gt;=0</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>1</td>
<td>activate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>Omega</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>2</td>
<td>Omega</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>3</td>
<td>Omega</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>4</td>
<td>Omega</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Since the transition synthesis method uses the method specifications to synthesize the transitions, it is limited to event-driven transitions (i.e., transitions that have associated events). For 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. For the NewAccount class example, we have identified two non-event-driven transition examples as shown in Table V. The invariant of state 3, which is different than the invariant of state 1, is "$\text{getUpdate()}\geq\text{MaxPeriod}\$". Therefore, this difference is added as a predicate to the non-event-driven transition that has the states 1 and 3 as source and destination states, respectively. The same situation applies for the transition that has the states 4 and 3 as source and destination states, respectively.

<table>
<thead>
<tr>
<th>Transition identifier</th>
<th>Source state identifier</th>
<th>Destination state identifier</th>
<th>Transition predicates</th>
<th>Transition actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>1</td>
<td>3</td>
<td>$\text{getUpdate()}\geq\text{MaxPeriod}$</td>
<td>-</td>
</tr>
<tr>
<td>21</td>
<td>4</td>
<td>3</td>
<td>$\text{getUpdate()}\geq\text{MaxPeriod}$ &amp; $\neg\text{!frozen}$</td>
<td>-</td>
</tr>
</tbody>
</table>

Moreover, using the proposed synthesis technique the time required to verify the correctness of the model is eliminated. The proposed technique does not guarantee synthesizing a free of infeasible paths model. Infeasible paths are the ones that cannot be executed. To solve this problem, we have to either detect the infeasible paths and avoid using them in generating the test drivers [17], or we have to ignore any test driver that has violated pre-conditions [3]. The proposed technique focuses on modeling classes that have sequential behaviors. Further research is required to model classes that have concurrent behaviors. To model such classes, synchronization contracts [18] can be used. A prototype tool is developed to automate the FIC testing process shown in Fig. 1. However, the first step of the process is not implemented yet. Further work is required to extend the tool to fully automate the testing process.

REFERENCES


