Automated Evaluation of Runtime Object States Against Model-Level States for State-Based Test Execution

Weifeng Xu  
Department of Computer & Information Science  
Gannon University  
Erie, PA 16541  
xu001@gannon.edu

Dianxiang Xu  
Department of Computer Science  
North Dakota State University  
Fargo, ND 58105  
dianxiang.xu@ndsu.edu

Abstract—Evaluation of runtime object states against the model-level states defined in a state model is critical to state-based test automation. This paper presents a state mapping framework to support the automated state-based test execution process. The framework automatically keeps track of runtime object states and maps these states to model-level abstract states. It also includes a comparator to determine whether or not a test is successful. This framework is implemented in the aspect-oriented programming language AspectJ. Therefore, the runtime instrumentation mechanism for state evaluation does not modify the source code under test (either in Java or AspectJ). This is an important advantage because otherwise all the changes made to the code for testing purposes have to be cancelled after testing. We have conducted several experiments to evaluate the framework by calculating and comparing the total consumed minutes of mapping states and checking the oracle in terms of manual and automated execution. The experiment results show that the mapping framework is much more effective and efficient than manual conversion.

Keywords: Software Engineering, software testing, model-based testing, state machine, state mapping.

I. INTRODUCTION

Software testing is expensive. Test case generation and test execution are two main activities in the software testing process. The cost of software testing can be reduced if they are automated. One way to automated generation of test cases is model-based testing Test execution is performed after the test cases are generated. Testers need to control the test case execution process, compare the actual outcomes to the expected results, and produce the execution report. For example, a state-based testing approach requires comparing actual states of running objects with the expected states in state modes to determine if a test is successful. Test execution is a tedious and time consuming process. It is often repeated from time to time. Test execution can be carried out more efficiently if it is automated.

A critical issue of state-based testing is to map runtime object states to the abstract states defined in a state model. Without such a mapping process, it would be difficult to automate state-based test execution. To address this issue, this paper presents a state mapping framework to support state-based test automation. The state mapping framework is able to automatically indicate whether or not an implementation under test (IUT) conforms to its state behavioral models (i.e., test oracle [1][11]). The challenges pertaining to the construction of such a state mapping framework include the following:

1. How does the mapping framework monitor and collect the states of running objects?
2. How does the monitor device interact with yet not depend on a particular IUT?
3. How can we automatically map the runtime object states to abstract states in a state model?
4. How can the test driver get informed if the actual state and the expected state do not match?

It’s worth mentioning that the input of the framework is the test cases that were generated by UML protocol state machines [7]. The output of the framework is the result of checking the test oracle for each test case. Note that any test execution framework, such as JUnit, can be used for checking the test oracle here after the concrete states are converted to abstract states.

Our approach starts with constructing a component that extracts expected states of the test cases generated by state models [7]. We also need a component for monitoring the properties of the running objects by exploiting the pointcut mechanism provided by aspect-oriented programming language AspectJ. This component converts the properties to a corresponding abstract state in an IUT’s state model. A third component is needed to report a failure message when actual objects states are determined to be different from the expected states.

The rest of this paper is organized as follows: section 2 reviews related work, section 3 describes the mapping process, section 4 presents a case study to demonstrate how the framework works, section 5 describes the experiments for evaluating the framework, and section 6 concludes this paper.

II. RELATED WORK

Our work is related to the state-based test case generation, JavaBeans, aspect-oriented programming and the test execution.
Xu et al. [2][7] and Tsai et al. [5] have proposed a state-based approach to systematically describe the behavior of the objects with state models, and then generate test cases from the state models. Their approaches use algorithms to generate tree-like structures (i.e., transition trees) from state models, where each path from the root of a transition tree to a leaf is a test sequence, i.e., an abstract test case. Concrete test cases are derived from the sequence in terms of test inputs. Xu et al.’s approach mainly focuses on detecting aspect-oriented faults as well as faults in object-oriented programs. They have developed a Model-based Aspect/Class Checking and Testing (MACT) framework to automatically generate test suites based on finite state machines [2].

Offutt et al. [9] have presented general criteria for generating test inputs from state-based specifications. The criteria include techniques for generating tests at several levels of abstraction for transition predicates, transitions, pairs of transitions, and sequences of transition. Boyapati et al. [6] have presented a framework to generate test case input for a method based on Java predicate and a bound on the size of its inputs.

Xie and Notkin [13] have developed the observer abstraction approach for automatically extracting object-state-transition information of a class from unit-test executions, without requiring a priori specifications. Their approach facilities programmers practically inspect the execution of each automatically generated test without a priori specifications. Their approach is able to produce the abstract state of an object based on the return values of a set of observers (public methods with non-void returns) invoked on the object. Comparing with our approach, the abstract states they have obtained are indirectly generated from a set of its initial tests generated by a third-party tool.

Aspect-Oriented Programming (AOP) [1] is an advanced way of modularizing crosscutting concerns, typically on top of OOP. AspectJ is an aspect-oriented extension created at PARC for the Java programming language. The elements to realize the crosscutting concerns in AOP include aspect, advice, pointcut and join point. An aspect is a separated module that expresses the crosscutting concerns. An advice expresses behaviors or responsibilities of the aspect. A pointcut defines a group of join points, and join points are the specific places in base code where advice can be dynamically merged into.

JavaBeans technology is the component architecture for the Java 2 Platform, Standard Edition (J2SE). Components (JavaBeans) are reusable software programs that you can develop and assemble easily to create sophisticated applications [10]. A JavaBean is able to observe changes to a collection of central business entities. A change event will be fired by the JavaBean, and then the event will be handled by an event handler (i.e., method propertyChange() ) in other components or business entities. The characteristics of a bean can be implemented in AOP. A Java class becomes a bean if the class is bound up with event handlers. The aspect is named a “BoundState”/ “BoundPoint” if it equips State/Point class with event handlers [2].

Our goal is to design an automated state mapping device to convert concrete states to their corresponding abstract states. Our work focuses on addressing the state conversion issue for facilitating the state-based test execution. We take Xu et al.’s generated test suites as the input for our state mapping framework. We take advantage of the pointcut mechanism of AspectJ to access the state of running objects, and then check the test oracle by utilizing the property of JavaBeans. We do not address how to derive concrete test cases from a test sequence based on test case generation criteria. Finally, we have integrated the framework as a component into MACT framework. The executable project can be accessed in http://www.cs.ndsu.nodak.edu/~dxu/research/MBT.html.

With extensions, the framework may be used with the other test generation approaches [6] [9].

III. STATE MAPPING INFRASTRUCTURE

This section presents the layered state mapping architecture, and then describes each of the components.

A. The Layered Structure

The layered state mapping infrastructure is shown in Figure 2. The infrastructure mainly contains two layers, i.e., the mapping framework layer and the user definition layer. We have included the IUT in the figure to facilitate our discussion.

![Figure 1. The layered test execution infrastructure](image)

The framework consists of the functionalities required for automated state mapping. The user defined layer extends the functionalities from the framework and defines other necessary information to meet specific needs for different IUT. The framework does not directly interact with IUT. It relies on the middle layer, i.e., user definition layer, to provide information for test execution. The middle layer reduces the coupling between the framework and the
IUT, and thus the changing of different IUT does not require any changes to the framework.

The static view of the state mapping framework is shown in Figure 2. We attach small circles to classes to indicate those classes have crosscutting properties, i.e., join points have been picked up in the classes by pointcuts. Specifically, those join points are specific places in the IUT where we want to observe the status of running objects. They are picked up by pointcuts in the aspects, i.e., AbstractBoundState and BoundState. Those aspects are denoted as dashed rectangles.

![Diagram of the state mapping framework](image)

Figure 2. The mapping infrastructure class diagram

### B. State Mapping Framework

The responsibility of the state mapping framework includes

- extracting expected states from test cases
- monitoring the properties of running objects
- mapping properties to high level abstract states, i.e., actual states
- comparing expected states with actual states, i.e., test oracle
- prompting error messages if expected and actual states are not matching

The mapping framework mainly consisted of two classes, StateBean and ModelTester, and an abstract aspect AbstractBoundState. The class ModelTester implements PropertyChangeListener interface.

Any state-based testing approach needs to have temporary space to store both expected and actual abstract states. The space is provided by the class StateBean. Moreover, the difference between those two states needs to be reported by the execution device. The report ability is the cooperated result of the aspect AbstractBoundState and the class ModelTester.

The class StateBean stores abstract states temporarily for each test case during test case execution. The class ModelTester is a property change listener. The class StateBean is integrated to the abstract AbstractBoundState aspect in order to form a bean class. The properties of the bean class (i.e., the name of a field), known as bound properties, generate and fire events whenever their states (values) change so that any registered property change listeners (i.e., other beans) will be informed of those changes. More specifically, we define a property of the StateBean as modelState, and it stores an expected state for each test case, where the expected state is derived from the condensed test oracle. The test oracle follows the format “assertState(obj, expState)”. The parameter obj represents an IUT, and the parameter expState represent the expected state of the IUT.

How does the state mapping device keep track of the current state of the running objects? During the test case execution, the abstract aspect AbstractBoundState is responsible for monitoring the current state of IUT. We exploit the pointcut mechanism of AspectJ to implement the monitoring ability of the state mapping framework. We first create an abstract pointcut checkpoints() to pick up specific places (i.e., join points) in IUT to be monitored. The checkCurrentState() method is invoked when those join points are reached during test execution. The pointcut is abstract because those join points are various in terms of different IUT. We will implement the pointcut in a subaspect.

Finally, the framework needs to report the result of the testing oracle. The results of the current state of IUT are sent to the StateBean, and are compared to the stored expected state. If the StateBean expected state is different from the actual state, a property change event will be generated and fired. The property change event fired by the StateBean will be handled by an event handler (i.e., method propertyChange() ) in class ModelTester. The class ModelTester is a property change listener that listens to the events sent from the StateBean, and it implements PropertyChangeListener interface. An appropriate message can be placed in the handler to provide a warning message if test cases fail.

### C. User definition layer

All specific information for testing individual IUT is defined in the user definition layer. The information is provided by users. It is either automatically generated by the test case generation process or manually defined by users. The user definition layer consists of two classes (i.e.,
IUTModelState and IUTModelTester and one aspect (i.e., BoundState). Those two classes can be automatically generated from state models. The aspect is the only component that needs to be manually implemented, and it picks up specific places in IUT to access the state of running objects. For example, after each function is invoked, the mapping device is required to check the states of the IUT.

To define specific check points in IUT, the BoundState aspect implements the abstract pointcut, (i.e., checkpoints) in its parent aspect (i.e., AbstractBoundState). It only allows the BoundState to communicate with the IUT so that replacing the IUT with a different one does not propagate side effects to the framework. In other words, it keeps the test mapping framework stable so that we can replace each IUT easily.

Which class maps the state of objects to abstract states defined in state models? The wrapper class IUTModelState is responsible for mapping physical states to abstract states. More specifically, we have implemented a converter, i.e., getModelState(), in the wrapper class to simply convert a set of properties of an IUT to a state space based on state predicates. For example, suppose a state predicate about a BankAccount class follows the format: “OPEN, !getClosed() && getBalance() >= 0 && !getFrozen()”. The semantic of the predicate indicates that the current state of a BankAccount object is OPEN if the bank account has a non-negative balance and the current state is neither CLOSED nor FROZEN. The predicates allow the converter to easily map physical states to abstract states. The wrapper class can be automatically generated based on Xu et al’s state models [7].

In addition, the wrapper class IUTModelState is named differently based on each specific IUT. According to the naming conversion, the three leading letters IUT are replaceable by the specific name of the IUT. For example, if the IUT is a class BankAccount, the wrapper class of the BankAccount is automatically named as BankAccountModelState. However, the name of its converter is fixed so that its client is able to invoke it without remembering the different name of the converter. The converter keeps the same signature as getModelState(). The association isolates the converter from the specific IUT.

Technically, to invoke the converter of a wrapper class, we take advantage of the pointcut mechanism to identify the running IUT, and then we exploit Java reflection to associate the IUT to its wrapper class and to invoke the state converter in the wrapper class. We describe the technical details as follows.

- **Step 1:** Exposing the context of the pointcut. The pointcut BoundState defines join points on IUT. The pointcut not only picks up join points, it also exposes part of the execution context at their join points [1]. The execution context tells the state monitor which object is running.

- **Step 2:** Capturing the value of state invariants. The premise to get the runtime states is that each variable in the IUT needs an accessor (i.e., getX()) so that the physical state of the running objects can be determined. The symbol “X” represents a property of IUT. It is not difficult to create accessors for class instance variables because lots of IDE can generate accessors, such as IntelliJ, NetBeans and Eclipse.

- **Step 3:** Converting physical states to abstract states. For each runtime object obtained in step 1, we use naming convention and Java reflection to create an object of wrapper class1. For example, for the IUT BankAccount, we first create the wrapper class named BankAccountModelState by invoking the following reflection method:

```java
Class.forName
(BankAccount.getSimpleName()+ "ModelState")
```

where the string “ModelState” is the postfix for each IUT. The converter can be created and invoked by a similar reflection method. During the execution of the state converter function, the accessor we mentioned in step 2 will be used. The resultant abstract states calculated by the converter will be used in the test oracle.

Finally, we need to create a test harness to invoke each test case. The class IUTModelTester consists of a collection of test cases derived from an IUT. It extends ModelTester in the mapping framework. Each test case contains a test oracle that overrides the method assert(obj, expState). Similar to the user defined class IUTModelState, the class IUTModelTester will be named differently for each IUT. For example, we will pack all the test cases in BankAccountModelState for testing BankAccount class. The test cases can be generated based on Xu et al’s algorithm [7].

**IV. CASE STUDY**

This section presents a case study to demonstrate how the state mapping framework automates the mapping process for the conformance testing.

We use BankAccount class as our benchmark to facilitate our discussion in the rest of the section. Furthermore, we use the test suites and wrapper classes that are generated from MACT framework [2] as our test execution input. The generated test suites and wrapper classes are written in Java.

Based on the state model of class BankAccount shown in the Figure 3, our framework generates two classes, BankAccountModelTester and BankAccountModelState.

---

1 Recent MACT versions 1.22 use an aspect instead of Java reflection.
The generated state wrapper class BankAccountModelState is shown in Figure 6. The state converter takes the runtime object as its input, and evaluates the current abstract states based on the accessors of the object. The following state predicates information is used for generating the wrapper state class: (OPEN, !getClosed() && getBalance() >= 0 && !getFrozen()), (FROZEN, getFrozen()==true) and (CLOSED,getClosed()==true). All states appeared in predicates are transformed into constant state variables and each expression is converted into corresponding if-else in the state converter. We omit the convert algorithm here because of the simplicity of the algorithm.

The test harness class BankAccountModelTester may contain a variety of test cases in terms of different testing coverage. For the demonstration purpose, we only list one test case example in Figure 4. Although we are not taking into consideration generating test suites, it is worth mentioning that Xu et al’s algorithm [7] also generated negative tests so after the BankAccount reaches the state CLOSED, any transitions from the CLOSED state will not be changed.

The test oracle assertState(bankAccount, BankAccount.OPEN) works as follows: it attempts to update a property of the StateBean object, i.e., modeState, that stores the runtime object state of bankAccount with the current expected state BankAccount.OPEN. If the runtime object state is different from the expected one, an event of state change will be generated. This indicates that the test case has failed. The event will be handled by the handler, i.e., propertyChange in ModelTester. An appropriate warning message will be sent by the event handler.

We use state predicates information to generate automatically the state wrapper class. The format of the state predicates is listed in Figure 5.

The advice pseudo code of the state JavaBean is listed in Figure 7. The setter will be invoked after each assertion method. The after device fires a property change event if the expected and actual states are not matching up. The event won’t be fired up if those two states are the same.

V. EXPERIMENTS

To evaluate our framework, we have conducted a series of experiments based on three applications: a telecom simulation, a cruise control simulation, and a banking system.

The telecom is a benchmark example available with AspectJ distribution [1]. It simulates how to record the connection time. The recording mechanism is simple - starting a timer when a connection circle is connected, and stopping the timer when the connection circle is disconnected. The cruise control simulation is a simulation system that automatically controls the speed of an automobile. The driver sets the speed and the system will take over the throttle of the car to maintain the same speed [8]. The banking system as a practical application has been used by researchers for different purposes [12].

The state model of class BankAccount

Figure 3. The state model of class BankAccount

The test harness class BankAccountModelTester may contain a variety of test cases in terms of different testing coverage. For the demonstration purpose, we only list one test case example in Figure 4. Although we are not taking into consideration generating test suites, it is worth mentioning that Xu et al’s algorithm [7] also generated negative tests so after the BankAccount reaches the state CLOSED, any transitions from the CLOSED state will not be changed.

The generated state wrapper class BankAccountModelState is shown in Figure 6. The state converter takes the runtime object as its input, and evaluates the current abstract states based on the accessors of the object. The following state predicates information is used for generating the wrapper state class: (OPEN, !getClosed() && getBalance() >= 0 && !getFrozen()), (FROZEN, getFrozen()==true) and (CLOSED,getClosed()==true). All states appeared in predicates are transformed into constant state variables and each expression is converted into corresponding if-else in the state converter. We omit the convert algorithm here because of the simplicity of the algorithm.

The test oracle assertState(bankAccount, BankAccount.OPEN) works as follows: it attempts to update a property of the StateBean object, i.e., modeState, that stores the runtime object state of bankAccount with the current expected state BankAccount.OPEN. If the runtime object state is different from the expected one, an event of state change will be generated. This indicates that the test case has failed. The event will be handled by the handler, i.e., propertyChange in ModelTester. An appropriate warning message will be sent by the event handler.

We use state predicates information to generate automatically the state wrapper class. The format of the state predicates is listed in Figure 5.

The advice pseudo code of the state JavaBean is listed in Figure 7. The setter will be invoked after each assertion method. The after device fires a property change event if the expected and actual states are not matching up. The event won’t be fired up if those two states are the same.

V. EXPERIMENTS

To evaluate our framework, we have conducted a series of experiments based on three applications: a telecom simulation, a cruise control simulation, and a banking system.

The telecom is a benchmark example available with AspectJ distribution [1]. It simulates how to record the connection time. The recording mechanism is simple - starting a timer when a connection circle is connected, and stopping the timer when the connection circle is disconnected. The cruise control simulation is a simulation system that automatically controls the speed of an automobile. The driver sets the speed and the system will take over the throttle of the car to maintain the same speed [8]. The banking system as a practical application has been used by researchers for different purposes [12].
```java
pointcut setter(State s):
    call(void State.set(*)) && target(s);
/**
 * Advice to get the property change event fired when the
 * setters are called. It's around advice because you need the old
 * value of the property
 */

after(State s): setter(s) {
    //get the name of the property. Each state is stored in one
    //property
    String propertyName = thisJoinPointStaticPart.
        getSignature().getName().substring("set".length());

    //get expected states
    //get current states based on the name of the expected states
    String expectedState = s.getState();
    String currentState = (String) ht.get(expectedState.substring(0,
        expectedState.indexOf(".")));

    //fire the event if those states are different
    firePropertyChange(s, propertyName,
        currentState, expectedState);
}
```

Figure 7. The pseudo code of a JavaBean fires events if the expected and
actual states are different

As shown in Table 1, we have selected six classes
among three projects as our experiments. A different
number of mutants have been implanted into each class,
and there is at least one mutant in each independent path
of the class. A number of tests are generated based on Xu et
al’s algorithm [7] . We use the generated test suites as
inputs for our state mapping framework in terms of classes.
Note that although we have listed the number of
the mutants in the table, we do not take consideration of the
type of the mutants into the framework evaluation.

Table 1. The number of tests as input for state mapping framework

<table>
<thead>
<tr>
<th>Application</th>
<th>Testing Class</th>
<th># of Test Methods</th>
<th>Number of Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Telecom</td>
<td>Connection</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Timer</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Cruise Control</td>
<td>CarSimulator</td>
<td>19</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Controller</td>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>SpeedControl</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>Banking</td>
<td>AccountJDBCImpl</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

For the same test suites, our experiments are designed to
answer two questions: (1) Is the state mapping framework
more efficient than manual mapping process during the test
execution? (2) How efficient is the state mapping framework
compared to the manual mapping process during the test execution?

For comparision purposes, we need to execute the same
tests with and without the state mapping framework.

To compare the efficiency of both mapping approaches,
we have divided 20 CS students in a software testing course
(Spring 2008) into two groups. Each group consists of 10
students. Students in group 1 are manually mapping the
states during the test execution, and students in group 2 use
the state mapping framework. We ask students to record the
time they spend on each of the test execution activities and
test results. The mapping activities mainly include the time
spent on monitoring/collecting the state of running objects
and converting states/checking test oracle. The execution
results are recorded in Table 2. Note that the time (in
minutes) consumed by the second group for collecting and
converting states is omitted.

Table 2. Comparing efficiency (in minutes)

<table>
<thead>
<tr>
<th>Testing Class</th>
<th># of states</th>
<th># of tests</th>
<th>Collecting</th>
<th>Conversion</th>
<th>Total</th>
<th>Group 1</th>
<th>Group 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Timer</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>7</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CarSimulator</td>
<td>6</td>
<td>19</td>
<td>30</td>
<td>19</td>
<td>49</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Controller</td>
<td>4</td>
<td>25</td>
<td>26</td>
<td>14</td>
<td>40</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>SpeedControl</td>
<td>4</td>
<td>9</td>
<td>12</td>
<td>7</td>
<td>19</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>AccountJDBCImpl</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>9</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>22</td>
<td>64</td>
<td>83</td>
<td>49</td>
<td>132</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

- Without any surprises, using the state mapping
  framework during the test execution is much more
  productive than without the framework. All the
test cases facilitated with the state mapping
  framework are completed in a couple of minutes
  (see Figure 8).

- Generally, the time spent on monitoring and
  collecting the state of the running object is (83-
  49)/49=69% more than converting the states from
  concrete to abstract (see Figure 9).
There is a directly proportional relationship between the number of states and the time consumed to map the states for the first group. The more states/predicates the IUT contains, the more time we have to consume.

Furthermore, mapping states without the mapping framework is error-prone. For example, we have found 4 out of 64 tests are missed or false alarmed. The missed test indicates test cases that should result in reporting mutants but have failed to do so, and the false alarmed means test cases that have triggered a false alarm. The average tests missed/false alarmed means the total number of tests missed and false alarmed tests is divided by the total number of students in the group.

VI. CONCLUSIONS

We have presented a framework for mapping the concrete states to abstract states for state-based test automation. The framework can keep track of runtime object states, convert them to the corresponding model-level states, and determine whether or not the actual states match the expectations. The framework is in essence an extension of the observer design pattern implemented by enterprise JavaBeans. However, we take advantage of the pointcut mechanism of AspectJ to facilitate state monitoring. The key benefit is that it does not modify the IUT. Our experimental results show that the mapping framework has greatly reduced the effort of state-based test execution. The more we re-ran the tests, the more we appreciated the state-mapping framework.

Regarding the future work, we plan to relax the assumption that each state-related variable has an accessor (i.e., getXXX() method) in the IUT. A possible approach is to use additional aspects as implicit implementation of all necessary accessors. There are other issues worth exploring as well. For example, we should remove check points manually defined in the BoundState aspect, and thus the user definition layer can be simplified. We have considered state abstractions, but not yet action abstractions (input/methods to call).

REFERENCES