Separation of Transitions, Actions, and Exceptions in Model-based Testing

Cyrille Artho

Research Center for Information Security (RCIS), AIST, Tokyo, Japan

Abstract. Model-based testing generates test cases from a high-level model. Current models employ extensions to finite-state machines. This work proposes a separation of transitions in the model and their corresponding actions in the target implementation, and also includes special treatment of exceptional states.

1 Introduction

Software testing entails execution of a system under test (SUT) or parts thereof. A series of stimuli (inputs) is fed to the SUT, which responds with a series of observable events (outputs). The oldest form of testing consists of executing a system manually, with a human administering the input and observing the output. In the last few decades, various techniques have been developed to automate testing, and to impose some rigor by systematic selection of test data and measurement of the effectiveness of tests [5,6].

For testing on a smaller scale, unit testing has become a widely accepted and used way of testing relatively small, self-contained units in software [4]. Unit testing automates test execution and the verification of the test output. In this way, once a test is written, it can be re-used throughout the life time of a system. The continuous use of a test suite throughout a product cycle is called *regression testing.* Unit tests automate regression testing, provided the interface (input specification) or the output format of a system does not change.

Despite its success, unit testing has the drawbacks that test creation often involves writing large parts of low-level code. Furthermore, unit tests require maintenance whenever the interface or output of the SUT undergoes a change. It is therefore desirable to automate test creation, while providing a more abstract view of inputs and outputs. The aspiration is to reduce the amount of low-level test code to be written, while also decoupling test behavior from the detailed format of data. In this way, the data format can be updated while the test model can be (mostly) retained.

Model-based testing (MBT) is a technology that automates creation of test cases in certain domains [3,8]. Test code is generated from the model by specialized tools [8] or by leveraging model transformation tools [3] from the domain of model-driven architecture (MDA) [7]. MDA represents an approach where problem-specific features (such as the description of system behavior) are represented by a domain-specific language. Standardized tools then transform the domain-specific language into the desired target format, such that an executable system is obtained. In this paper, the domain-specific language entails a description of the system behavior as a finite-state machine (FSM). This description is then transformed into Java code.

Prior to this work, to our knowledge, openly available MBT tools for Java lacked the flexibility of the approach that is presented here. This paper makes the following contributions:

1. The existing test model used in ModelJUnit [8] is replaced with a high-level model that minimizes the amount of code to be written, while being more flexible.
2. Our new model separates two orthogonal concerns, model transitions and implementation actions, presenting a conceptually cleaner solution.
3. Exceptional behavior, a feature that is often used in modern programming languages, can be modeled naturally and expediently with our architecture.

The rest of this paper is organized as follows: Section 2 introduces a running example that shows how the model handles different aspects of the problem. Problems with the existing approach are detailed in Section 3. Our proposed architecture is described in Section 4. Section 5 concludes and outlines future work.

2 Example

2.1 Elevator system

The example that is used throughout this paper describes the possible state space of an elevator that has two pairs of doors: one at the front, another one at the rear. The elevator has been inspired by an elevator from Yoyogi station of the Oedo metro subway line in Tokyo. The elevator ranges over five floors, from the street-level entrance on the second floor, to the lower entrance on the first floor (reachable through a passage from a different train station), to the underground floors of the actual metro station. While the first and second basement level exist, they are not reachable by the elevator, and in general inaccessible to subway commuters, even by stairs.[1]

The model has to reflect the range of permitted configurations and operations that change the system state. Figure 1 shows the state space of all permitted elevator states. Each state has its own label, while transitions are labeled with the action required to reach the next state from the current state. This example does not include transitions leading to error states; such an addition will be described later.

2.2 Test cases using JUnit

A simple property that has to hold for the given elevator example is that a if it moves down a number of floors, it will be back at the starting position after having moved up again the same number of floors. If the doors open and close in between, the property still has to hold. A few unit tests verifying this property are shown in Figure 2. As can

[1] Construction of the Toei Oedo line started in 1991, when most of Tokyo city and its metro network were already built. This made it necessary to construct the Oedo line more than 40 m below ground, requiring the elevator to skip a couple of levels before arriving at the ticket gate on the third underground floor. On that floor, the rear doors open. This design makes it more convenient to use the elevator, as people to not have to back out of the elevator through the same door they entered in, but they can instead use the opposite door in front of them.

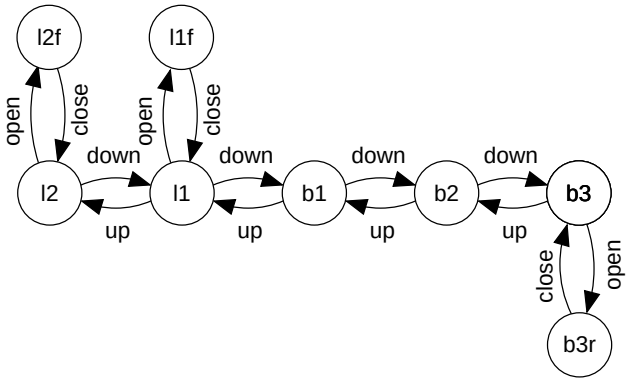

Fig. 1. Example of an elevator with two doors, which are only allowed to open on certain floors. On levels 1 and 2 (states l_1 and l_2), the front doors are allowed to open, changing the configuration to state l_{1f} and l_{2f}, respectively. On the first and second basement floor (b_1 and b_2), the doors must remain closed. On the third basement floor, the rear doors are allowed to open, which is reflected by transition $b_3 \rightarrow b_{3r}$.

be easily seen, much of the test code is simple and repetitive. Yet, only a small part of the possible state space is covered.[2] While better coverage can be achieved in principle, it is unlikely to happen in practice, as too much code has to be written. Model-based testing aims to replace the manual work required to generate a high coverage of the possible state space, while keeping the task of modeling the system simple.

```
@Test void test1() {
  pos = l2;
  down();
  up();
  assert(pos == l2);
}

@Test void test2() {
  pos = l2;
  down();
  open();
  close();
  up();
  assert(pos == l2);
}
```

```
@Test void test3() {
  pos = l2;
  down();
  down();
  down();
  down();
  open();
  close();
  up();
  up();
  up();
  up();
  assert(pos == l2);
}
```

Fig. 2. Code example of unit tests using JUnit.

[2] For instance, the possibility of moving to the bottom floor without opening the doors is not tested; opening and closing the doors twice before moving on is not tested; and opening the doors on the first floor before moving to the bottom floor is not tested either.

3 MBT Using ModelJUnit

3.1 Tool architecture

ModelJUnit is an openly available test case generation tool [8], using an extended finite state machine (EFSM) as input. An EFSM is a Java implementation of a finite-state machine, including two extensions: First, a custom state comparison function can provide a more refined or more abstract behavior than a pure FSM. Second, transitions of an EFSM include their mapping to concrete actions of the system under test (SUT).

ModelJUnit searches the graph of an EFSM at run-time. As new transitions are explored, corresponding actions in the SUT are executed. Results are verified by using JUnit assertions [4]. Specification of the entire model as a Java program allows one artifact to describe all system states, transitions, and corresponding actions of the SUT. However, states and transitions have to be encoded in Java. Each transition requires two methods: a *guard* specifying when a transition is enabled, and an *action* method specifying the successor state and the corresponding action of the SUT. In ModelJUnit, each transition/action pair requires about six lines of repetitive yet error-prone code.

3.2 Elevator system described in ModelJUnit

As described above, ModelJUnit explores a specifically structured Java program. The program has to contain an initialization method, a number of guard conditions, and actions that update the model state and execute test actions. The tasks of exploring the model, measuring and reporting model coverage, are then done by ModelJUnit. Figure 3 shows an excerpt of a finite-state machine encoded for ModelJUnit. As can be seen, the specification of the state space and the initial state is rather simple, but parts relating to the SUT (lines 2 and 8) are interleaved with model code. This interleaving of model and implementation code continues throughout each guard method (lines 12–14) and action method (lines 16–19). Out of these seven lines, only one line, the call to `down()`, corresponds to a concrete test action. The remainder of the code is quite repetitive and can be fully expressed by the graphical finite-state machine from Figure 1.

4 Proposed Architecture

4.1 Separation of the FSM from system actions

We propose a separation of the behavioral model and the program code. The state space of our model is described by a conventional FSM, using the "dot" file format from graphviz [1]. This format is concise, human-readable, and supported by visualization and editing tools. A transition can be specified in a single line of text of form `pre -> post [ label = "action" ]`. A label corresponds to an action in the SUT. The same action may be executed in different transitions.[3]

Most methods of a SUT require arguments that cannot be constructed trivially from a high-level model such as an FSM. Instead, actions are delegated to a bridge class

[3] The same effect would be achieved in an existing ModelJUnit model by either duplicating code or by writing complex transition guards and post-state conditions.

```
 1 public class ElevatorFSM implements FsmModel {
     private ElevatorController elevator;

     enum State { l2, l2f, l1, l1f, b1, b2, b3, b3r }
 5   private State state;

     public void reset(boolean testing) {
       elevator = new ElevatorController();
       state = State.l2;
10   }

     public boolean ActionL2L1Guard() {
       return state == State.l2;
     }
15
     public @Action void actionL2L1() {
       elevator.down();
       state = State.l1;
     }
20 }
```

Fig. 3. Part of the elevator model for ModelJUnit.

written in Java. The bridge class implements test actions and describes how parameters are constructed and verified.

4.2 Elevator model using our architecture

Figure 4 shows how our model splits the concerns into two parts: The FSM, encoded in the dot format, and the bridge class called *ElevatorImpl*, which contains the application-specific test code. Note that the entire state space of the model is conveniently managed in the FSM, so our Java code contains no guard or state variables. At the same time, any implementation-specific code is removed from the FSM.

4.3 Model annotations

Libraries may contain redundant interfaces (“convenience methods”) as shorthands. It is desirable to test all of these methods, yet the FSM would be cluttered by the inclusion of the full set of redundant methods. We chose to use annotations of FSM transitions to describe cases where a single FSM transition covers a set of actions. Annotated transitions are internally expanded to the full set of methods before the test model is explored. Thereby, interface variants are tested by selecting a random variant each time the transition is executed. A set of methods that only read data without changing the system state can also be represented by one FSM action and an annotation. In this way, the fact that no access method actually modifies the system state can be tested against.

```
digraph Elevator {
  init -> l2;                      b1  -> l1  [ label = "up" ];
  l2  -> l1 [ label = "down" ];    l1  -> l2  [ label = "up" ];
  l1  -> b1 [ label = "down" ];    l2  -> l2f [ label = "open" ];
  b1  -> b2 [ label = "down" ];    l2f -> l2  [ label = "close" ];
  b2  -> b3 [ label = "down" ];    l1  -> l1f [ label = "open" ];
  b3  -> b2 [ label = "up" ];      l1f -> l1  [ label = "close" ];
  b2  -> b1 [ label = "up" ];      b3  -> b3r [ label = "open" ];
                                   b3r -> b3  [ label = "close" ]; }
```

```
public class ElevatorImpl {
  private ElevatorController elevator;

  public void init() {
    elevator = new ElevatorController();
  }

  public void down() {
    elevator.down();
  }
}
```

Fig. 4. The FSM of the elevator model in dot format (top) and a part of the bridge to its implementation in Java (bottom).

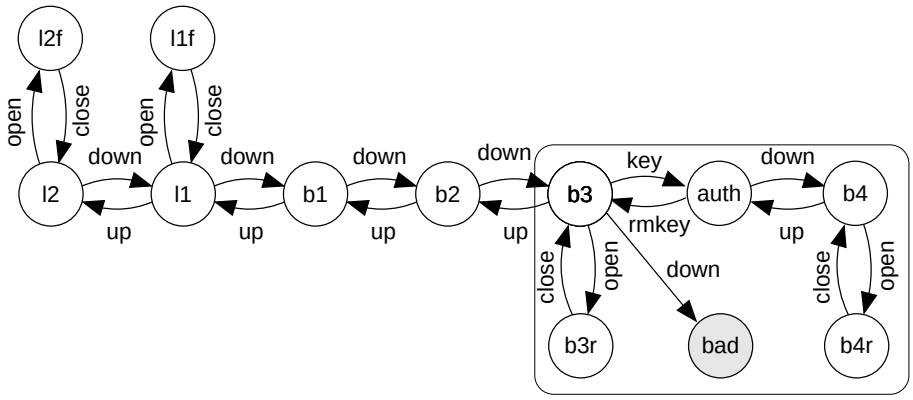

Fig. 5. Extended model of the elevator. Assume there is a forth basement level, which is only allowed to be accessed by authorized personnel. Authorization is granted when a key is inserted when in basement level 3 (b_3). If the "down" action is executed without authorization, an exception should be thrown, as indicated by state *bad*.

4.4 Exceptional behavior

Finally, a given action may cause an exception, depending on the state of the SUT. Exception annotations specify states where an exception is expected to occur. As an example, take an extension of the elevator from the ticket gate level (b_3) to the tracks (b_4). Customers are required to use the ticket gate for the metro, so they are not authorized to access level b_4 directly through the elevator. Assume that, for maintenance, the elevator extends to b_4. If a key is inserted in b_3, the elevator changes to an authorized state, from which access to the lowest level is given. Otherwise, any attempt to reach b_4 should be indicated by an exception, leading to a "bad" state.[4] Figure 5 shows the extended FSM.

The code necessary in Java to verify the presence or absence of an exception is quite lengthy. In Java, so-called *checked exceptions* are defined by methods that declare that they may throw an exception [2]. Each call to such a method has to be guarded against possible exceptions by using a `try`/`catch` block. In our model, annotations of FSM states specify which states correspond to an exceptional state. All transitions to that state are expected to throw precisely that type of exception. Equivalent actions leading to non-exceptional states may not throw any exception. As the two cases differ only slightly, the code in Figure 6 can be easily generated automatically.

```
public void actionb3bad() {
  boolean exceptionOccurred = false;
  try {
    impl.down();
  }
  catch (IllegalStateException e) {
    exceptionOccurred = true;
  }
  assert(exceptionOccurred);
}
```

Fig. 6. Code that verifies the presence of an exception.

4.5 Tool architecture

For use with ModelJUnit, the FSM is expanded into its Java representation (see Figure 7). The generated code includes `try`/`catch` blocks for verification of the presence or absence of exceptions. Transformation of the FSM can be automated either by leveraging existing model transformation tools [7] or by extending ModelJUnit with a parser for the additional file formats. The former approach requires additional tools but is independent of the programming language and unit test library.

[4] In the Tokyo Metro stations, this problem is usually solved by having a separate elevator from the ticket gates down to the tracks below.

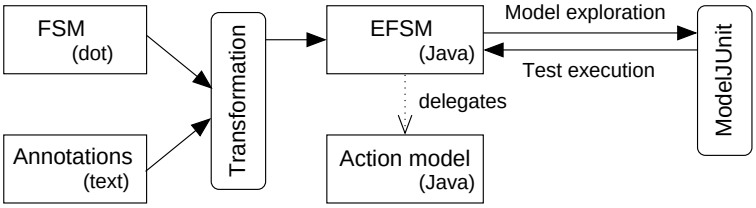

Fig. 7. Architecture of system to generate the ModelJUnit model.

5 Conclusions and Future Work

Current tools for model-based testing do not completely separate all features. Specifically, different transitions in an abstract model may correspond to the same action in the implementation. Separation of these two artifacts leads to a more concise model. Inclusion of exceptional states in the model further increases the amount of code that can be generated automatically, making the model more expressive and maintainable.

Future work includes exploring the possibility of tying state invariant code, rather than just exceptions, to states in the model. This would be done in a way that is equivalent to how actions are tied to transitions. Furthermore, in our architecture, the ModelJUnit tool only serves to explore the graph of the model. The transformation step shown in Figure 7 could therefore be subsumed by direct execution of the action model, without going through ModelJUnit. Independence of ModelJUnit would have the added benefit of being able to implement features that ModelJUnit does not support, such as non-deterministic transitions.

References

1. E. Gansner and S. North. An open graph visualization system and its applications. *Software – Practice and Experience*, 30:1203–1233, 1999.
2. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, 2005.
3. A. Javed, P. Strooper, and G. Watson. Automated generation of test cases using model-driven architecture. In *Proc. 2nd Int. Workshop on Automation of Software Test (AST 2007)*, page 3, Washington, USA, 2007. IEEE Computer Society.
4. J. Link and P. Fröhlich. *Unit Testing in Java: How Tests Drive the Code*. Morgan Kaufmann Publishers, Inc., 2003.
5. G. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., 1979.
6. D. Peled. *Software Reliability Methods*. Springer, 2001.
7. J. Poole. Model-driven architecture: Vision, standards and emerging technologies. In *Workshop on Metamodeling and Adaptive Object Models*, Budapest, Hungary, 2001.
8. M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, USA, 2006.