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

Unit testing has become a widespread in software quality assurance [3]. Model-based testing (MBT) allows for automated creation of test cases in certain domains [2,5]. Test code is generated from the model by specialized tools [5] or by leveraging model transformation tools [2] from the domain of model-driven architecture (MDA) [4].

2 MBT using ModelJUnit

ModelJUnit is an openly available test case generation tool [5], 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 [3]. 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 method specifying the successor state and the corresponding action of the SUT. In ModelJUnit, each transition/action pair requires about a dozen lines of repetitive yet error-prone code.

3 Proposed architecture

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.[1]

[1] 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.

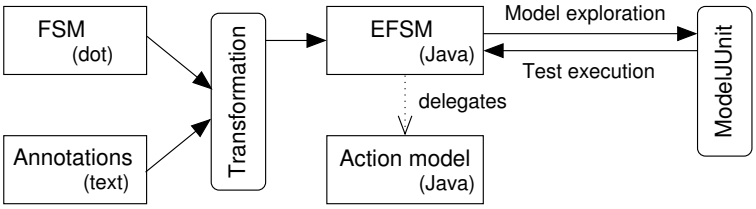

Fig. 1. Architecture of system to generate the ModelJUnit model.

Libraries may contain redundant interfaces as shorthands. Annotations of FSM transitions describe cases where a single FSM transition covers a set of actions. Interface variants are tested by selecting a random variant each time the transition is executed.

Most methods of a SUT require arguments that cannot be constructed trivially from a high-level model such as an FSM. We delegate actions to a separate Java class, which implements these actions and describes how parameters are constructed and verified.

Finally, a given action may produce an exception, depending on the state of the SUT. Exception annotations of FSM states specify states where an exception has occurred.

For use with ModelJUnit, the FSM is expanded into its Java representation (see Figure 1). The generated code includes a `try`/`catch` block for verification of the presence or absence of exceptions. Transformation of the FSM can be automated either by leveraging existing model transformation tools [4] 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 Conclusions

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.

References

1. E. Gansner and S. North. An open graph visualization system and its applications. *Software – Practice and Experience*, 30:1203–1233, 1999.
2. 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.
3. J. Link and P. Fröhlich. *Unit Testing in Java: How Tests Drive the Code*. Morgan Kaufmann Publishers, Inc., 2003.
4. J. Poole. Model-driven architecture: Vision, standards and emerging technologies. In *Workshop on Metamodeling and Adaptive Object Models*, Budapest, Hungary, 2001.
5. M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, USA, 2006.