Techniques for Improving Test-Driven Design

Martin Wirsing, Hubert Baumeister, and Alexander Knapp
LMU München, Institut für Informatik, Oettingenstr. 67, D-80638 München
Email: {wirsing, baumeist, knapp}@informatik.uni-muenchen.de

Abstract

Early test development and specification enhance the quality and robustness of software as experience with new agile software development methods shows. The methods propagate test-first techniques and early prototyping through executable design models. For UML, Model-Driven Architecture is oriented towards executable models. Several authors propose scenarios specified by sequence diagrams as test cases for state diagrams; more generally, using software model checking one may automatically verify whether state diagrams or code satisfy properties defined by sequence diagrams. Other approaches use OCL invariants and pre-/post-conditions for instrumenting Java code with assertions.

Also, Extreme Programming (XP) requires to write tests before writing the code as a means for making the software robust and more easily refactored. Popular tools for XP are the family of xUnit tools—with JUnit as the most well known instance—for writing automated unit tests and Fit for writing automated acceptance tests.

In this paper we propose techniques for extending and improving such test-driven development methods, where executable tests drive the development process. Scenarios and properties serve us as a combined basis for system specification and test cases. Scenarios are defined by sequence diagrams written in a powerful sublanguage of UML 2.0 which allows us to specify not only possible scenarios but also forbidden scenarios (failure traces). A forbidden scenario is a scenario where one wants to say that after legally performing some steps, the next step should now occur. This is not expressible UML 1.5 sequence diagrams. Further extensions w.r.t. other approaches are the use of nested method invocations and state invariants.

Scenarios are examples of successful or un-successful system runs. By extracting common properties of several scenarios we obtain invariants and pre-/post-conditions written in OCL or JML. The behaviour of the system is described either by models such as state diagrams or activity diagrams, or by code e.g. written in Java.

For testing we insert invariants and pre- and post-conditions as assertions in the code and the behaviour models. Then we test the instrumented system behaviour with respect to the possible and forbidden scenarios. This is done by translating possible and forbidden scenarios to Fit tests for scenarios involving user interaction and to JUnit tests for system scenarios.

Due to the addition of the assertions to the system behaviour we obtain a more complete test coverage and further possibilities for checking dynamically the internal consistency of the system specification.

For verification, we propose two approaches: interactive theorem proving combined with symbolic evaluation and model checking. To be successful with the latter technique we have to restrict the models to finite domains. Therefore we construct suitable abstractions of the scenarios and the system behaviour and verify the abstractions using a model checker. For verifying the general case, symbolic evaluation helps to reduce considerably the number of necessary interactions with an interactive theorem prover.

Currently we are integrating these techniques into a user-oriented collaborative development environment.