Model-Based Testing Using LSCs and S2A⋆,⋆⋆

Shahar Maoz1, Jani Metsä2,⋆⋆⋆, and Mika Katara2

1 The Weizmann Institute of Science, Rehovot, Israel
2 Tampere University of Technology, Department of Software Systems, Tampere, Finland

Abstract. We report on our preliminary experience in using high-level visual scenario-based models for tests specification, test generation, and aspect-based test execution, in the context of an industrial application. To specify scenario-based tests, we used a UML2-compliant variant of live sequence charts (LSC). To automatically generate testing code from the models, we used a modified version of the S2A Compiler, outputting AspectC++ code. Finally, to examine the results of the tests, we used the Tracer, a prototype tool for model-based trace visualization and exploration. Our experience reveals the advantages of integrating models into industrial settings, specifically for model-based test specification and aspect-based execution: generating aspect code from visual models enables exploiting the expressive power of aspects for testing without manual coding and without knowledge of their rather complex syntax and semantics. We further discuss technological and other barriers for the future successful integration of our initial work in industrial context.

1 Introduction

Model-based test techniques may provide benefits over conventional script-based test automation solutions in terms of productivity and test coverage. Still, at least two major challenges hinder the adoption of such testing approaches. First, difficult deployment and suboptimal use of technology due to testers lack of specialized modeling skills. Second, the use of technology that limits the high-level testing of the system under test (SUT) to interface testing, where inputs are passed as parameters and the output is observed only from the return values.

In this work we introduce a novel approach to model-based testing where models based on high-level visual scenarios are compiled automatically into test aspects. This aims at partly addressing the above challenges, using a visual language to make test specifications more accessible to engineers while taking advantage of aspect-oriented technology in order to access the SUT internals.

Specifically, to visually specify testing scenarios we use a UML2-compliant variant of Damm and Harel’s live sequence charts (LSCs) [1,2]. LSCs is a visual formalism

* Empirical results category paper.
** The first author would like to acknowledge partial funding from an Advanced Research Grant from the European Research Council (ERC) under the European Community’s 7th Framework Programme (FP7/2007-2013). The other authors acknowledge partial funding from the Academy of Finland (grant number 121012).
*** Part of this author’s work was done while he was with Nokia Corp., Devices R&D.

© Springer-Verlag Berlin Heidelberg 2009
that extends classical sequence diagrams partial order semantics mainly by adding universal and existential hot/cold modalities, allowing a visual and intuitive specification of scenario-based liveness and safety properties. To execute the tests, we automatically translate the diagrams into test scenario aspects using a modified version of the S2A compiler \cite{3}. After weaving with the SUT code, the generated aspects follow the execution of the tests specified in the diagrams and report on their run time progress and results using scenario-based traces \cite{4}, which are visualized and explored in a prototype tool called the Tracer \cite{5}.

Aspects have been used for testing before (see, e.g., \cite{6}). Our approach to using generated scenario aspects for test execution has a number of advantages. First, test definition is done visually, using popular standard diagrams, within a commercial tool, and does not involve code writing. Second, the diagrams are automatically translated into test scenario aspects, which are woven into the code of the SUT, taking advantage of aspect technology in order to access the SUT internals without explicitly changing the original code. Third, the results of the tests are not limited to Boolean pass/fail output but instead provide rich information on traces of execution, exactly at the level of abstraction defined by the scenarios used for testing.

We report on a preliminary case study where we tried out our approach on an industrial system: a C++ application, running on Symbian OS, inside a Nokia smartphone. The case study has been carried out by the second listed author while he was at Nokia Corp., Devices R&D. We present the study results, and further discuss technological and other barriers for the future successful integration of our work in the industrial context.

This short paper focuses only on the introduction of the tool chain and on the initial case study evaluation. An extended version that includes background material, example diagrams and code snippets from the case study, additional technical details on the aspect code generation, a discussion of related work etc., is available as a tech. report\cite{1}.

2 Overview of the Tool Chain and Case Study

Defining the Scenarios. First, the test designer draws LSCs (that is, UML2-compliant modal sequence diagrams) using IBM RSA \cite{7} (extended with the modal profile defined in \cite{2}). The profile extension allows the engineer to set hot and cold modes to methods and conditions, as required by LSCs. In general, any UML2-compliant editor that supports profiles could be used to draw the LSCs. A number of LSCs are drawn, divided between several use cases for better manageability.

Some of the scenarios monitor for forbidden behaviors. If they occur, a violation is recorded. Note that the modeled scenarios combine monitoring with execution; they do not only listen for relevant events to monitor the progress of the tests. Rather, some methods are designated with the execution mode. When such a method is enabled in one chart and not violating in any other chart, the generated code, described next, executes

\begin{footnotesize}

\url{http://practise.cs.tut.fi/publications.php?project=amoeba-testing}
\end{footnotesize}