

# Tool Support for Continuous Maintenance of State Machine Models in Program Code

Moritz Balz, Michael Striewe, Michael Goedicke  
Paluno – The Ruhr Institute for Software Technology  
University of Duisburg-Essen, Campus Essen, Germany  
{moritz.balz, michael.striewe, michael.goedicke}@s3.uni-due.de

## Categories and Subject Descriptors

D.2.2 [Software]: Design Tools and Techniques—*State diagrams*; D.2.4 [Software]: Software/Program Verification—*Validation*; D.3.3 [Software]: Language Constructs and Features—*Patterns*

## General Terms

Design, Languages, Verification

## ABSTRACT

Software under development is considered by developers at different levels of abstraction, often with formal model specifications that describe actual program code. However, there are semantic barriers between high-level specifications and the resulting programs. In this contribution we introduce a set of tools that maintain multiple abstraction levels in appropriate program code patterns throughout the development process, including run time and monitoring. This makes the program code the only notation necessary for expressing different abstraction levels and improves maintenance of high-level specifications, synchronization of different specifications, and design recovery.

## 1. INTRODUCTION

In previous publications we introduced the *embedded models* approach that represents model specifications in program code patterns of general-purpose programming languages [1]. By this means abstract model specifications are tightly coupled to implementations that are not based on formal models. The approach is appropriate in cases where large parts of the software are written manually due to frameworks, architectures, or special requirements which cannot easily be integrated into generated program code or executable model specifications. In such cases, program code with embedded models is interpretable at different levels of abstraction, so that specifications can be stored there and used to design the program, verify it against formal specifications, control its execution at run time, and monitor its run time behavior with respect to models.

The approach has so far been evaluated with the implementation of a non-trivial application for load generation in performance tests [4]. The strategies for load generation used in this application are modeled as state machines and

embedded into the surrounding source code of the application, which is not based on models, but on structures determined by certain frameworks in use. In this contribution we present a suite of tools that realize working with program code at different abstraction levels for the domain of state machine models. We describe the approach for modeling state machines and the architecture of our tool suite in section 2. The components for design, verification, and monitoring are introduced in sections 3 to 5. We afterwards describe expected benefits in section 6.

## 2. APPROACH

Embedded models define program code patterns that represent the abstract syntax of formal model specifications and are executable by means of reflection. Thus program code contains not only modeling information, but can at the same time be interpreted and invoked so that it fulfills the purpose of the underlying models at run time. The program code patterns are connected to other program code by interfaces that realize abstractions to arbitrary business logic. The program code pattern for state machines follows some basic rules, which are comprehensible in detail in our previous publications [1]: States are represented by class definitions; transitions are represented by methods in state classes; meta data annotations [3] on transition methods refer to target state classes. Transition methods contain sets of method calls that can be interpreted as action labels. Guards and updates are represented by methods containing expressions that are related to variables defined as methods in a designated interface. An execution framework interprets the static structures and invokes transitions, guards, and updates to determine the state machine flow.

The tool suite for embedded state machines consists of plugins for the Eclipse IDE. The single tools access program code with Eclipse's Java development tools or the reflection mechanisms of the Java platform. By this means modeling, verification, and monitoring of state machine specifications are integrated in a tool chain that works with the same artifacts used for manual development of program code.

## 3. DESIGN

One of the most-perceived benefits of abstract models are visualizations supporting design and comprehension. This also applies to state machines. The tool suite thus contains a visual editor for editing program code representing state machines. It allows developers to select an existing Java package containing source code related to state machines, or alternatively to create a new one. If source code exists,

the contained model is then displayed in the editor. States and transitions can be created, modified, and removed in the graphical view. This implies creation, modification, or deletion of classes and methods in the source code. The names of states and transitions are defined in the graphical view and influence the names of the classes and methods. These structures can be mapped unambiguously except layout information, which is not included in the program code pattern. To apply a layout, the editor must rely on heuristics; for the future it is planned to save layout information externally, which would be acceptable because it affects meta information only and not the actual model specifications.

Guards and updates for transitions are edited as expressions containing variable names, operators, and literal values. They are mapped to expressions in the program code containing calls to the variables interface, Java operators, and literal values. This is realized with Eclipse's Document Object Model (DOM) providing fine-grained access to the syntax tree of Java classes. Action labels are assigned to transitions with a selection from available action labels and the definition of an invocation order by the user. This is realized in the tool by considering the method names of interfaces to other program code. Available channels can also be assigned to transitions for sending or receiving.

All structures are unambiguous so that the related source code can be created or adjusted simultaneously when the model is saved. Since states, transitions, contracts, actions, variables, and channels can be identified by their names, the related code fragments are adapted if applicable. The editor does thus not rely on an external description of the model, but reads and modifies the source code defined by the embedded model only.

## 4. VERIFICATION

When state machines are embedded in program code, the resulting programs can be verified at the model level. Tool support for this is possible by integrating model checking capabilities into the editor described above. However, sophisticated tools for model checking of state machines already exist, so it is desirable to re-use them instead of developing new tools. We thus provide a tool that extracts modeling information from program code and exports it into the data format used by UPPAAL [2], which is (despite its capability to handle timed automata, which is not used for our purpose) able to check automata with respect to CTL formulas as well as to general automata properties like deadlocks.

With respect to automata used by UPPAAL, the state machine model elements are mapped to the UPPAAL notation. Since the verification is intended to focus on properties of abstract model specifications, low-level details of the programming language and algorithms in the business logic are disregarded. The export is realized as a graph transformation between the abstract syntax graph of the embedded state machine and the DOM tree of UPPAAL's data format. It considers information about states, transitions, variable definitions, guards, and updates. Action labels are not part of UPPAAL models and are therefore neglected. As a result of this mapping, the extracted model is fully decoupled from program code and thus self-contained.

## 5. MONITORING

With the design and verification tools, the development

of programs containing embedded state machines is covered. Considering the run time, a benefit of embedded models is that the modeling information is also available in compiled program code to a certain degree; in Java, classes, methods, and meta data annotations can be accessed with reflection. By this means monitoring with respect to abstract specifications is possible without relying on tracing meta data. For this purpose the execution framework for embedded state machines emits information when it interprets and invokes program code: Initialization and start of a state machine, activation of states, selection of transitions, and validation of updates after a transition.

The monitoring tool receives this information and shows a graphical representation of the state machine, highlighting active states and transitions. The state space is monitored with a list of the variables showing current values according to the last update as well as the previous values before this update. The user is in addition notified about updates that could not be validated successfully, i.e. when the state space is influenced by the run time data differently than predicted with the model's post conditions. Since the execution framework is in full control of the program flow, the views are accompanied by buttons to pause and resume state machine execution that can be operated when an event is handled.

## 6. EXPECTED BENEFITS

This tool suite is appropriate for developing programs whose source code is written manually, but based on state machine models, like the load generator application mentioned above. In this case modeling information and program code are tightly coupled, and information at different abstraction levels is available in a coherent notation. By this means the semantic gap between specifications and resulting programs is narrowed in an automated way supported by the tools, since the models serve not only as requirements or documentation, but are executed themselves. Compared to modeling tools based on code generation or model execution, no modeling languages must be used in addition to program code, so that the hurdle is lowered to use formal specifications in existing development infrastructures.

## 7. REFERENCES

- [1] M. Balz, M. Striewe, and M. Goedicke. Continuous Maintenance of Multiple Abstraction Levels in Program Code. In *Proceedings of the 2nd International Workshop on Future Trends of Model-Driven Development - FTMD 2010, Funchal, Portugal*, 2010.
- [2] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1-2):134-152, Oct 1997.
- [3] D. Schwarz. Peeking Inside the Box: Attribute-Oriented Programming with Java 1.5. *ONJava.com*, June 2004. <http://www.onjava.com/pub/a/onjava/2004/06/30/insidebox1.html>.
- [4] M. Striewe, M. Balz, and M. Goedicke. SyLaGen - An Extendable Tool Environment for Generating Load. In B. Müller-Clostermann, K. Ehtle, and E. Rathgeb, editors, *Proceedings of "Measurement, Modelling and Evaluation of Computing Systems" and "Dependability and Fault Tolerance" 2010, March 15 - 17, Essen, Germany*, volume 5987 of *LNCS*, pages 307-310. Springer, 2010.