Development of Model Based Tools to Support the Design of Railway Control Applications

István Majzik, Zoltán Micskei and Gergely Pintér

Department of Measurement and Information Systems
Budapest University of Technology and Economics, Budapest, Hungary
{majzik, micskeiz, pinterg}@mit.bme.hu

Abstract. The development standard for railway control software requires several design and verification methods. To support these methods we elaborated a coherent set of tools based on UML state diagrams. To avoid the problems of the ambiguous UML semantics, we propose a subset of UML state machines that includes the practical modeling concepts and has well-defined operational semantics elaborated definitely for software engineers. Based on this formalism we developed a tool chain supporting (i) the simulation of the behavior specified by the state diagram, (ii) static checking the completeness and consistency of the specification, (iii) generation of the C source of the application control flow, (iv) automatic construction of test cases on the basis of structural test coverage criteria and (v) automatic construction of the source code of run-time verification procedures that aim at checking high-level safety properties.

Keywords: UML state diagrams, static checking, assertions, test generation

1 Introduction

The software development standard EN 50128 for computerized railway control systems prescribes several methods and techniques that are mandatory or highly recommended at a given safety integrity level. Among others, we can mention static analysis, failure assertion programming and structural testing. Some of these methods can be supported by automatic tools. However, tool application in the design and verification phases needs a clear understanding of the formalisms and models that form the input of the tools and allow the interpretation of the results produced by these tools.

In the framework of a project supported by the Hungarian National Office of Research and Technology\(^1\) we have elaborated a coherent set of tools and techniques based on UML statecharts models. UML statecharts as a modeling language could be effectively used in the design of event-triggered state-based control systems, however, its use in safety critical applications was hindered by its ambiguous standard semantics (often reported in the literature) and usability problems appeared in connection with the formal semantics developed so far. Accordingly, we have elaborated a subset of UML 2.0 state machines called Precise Statecharts. It includes all practically

\(^1\) Project nr. GVOP - 3.1.1 - 2004 - 05 - 0523/3.0
meaningful modeling concepts and has a fully defined operational semantics that was elaborated definitively for software engineers (instead of computer scientists). This formalization step allowed the development of the following support tools: (i) simulation of the control flow specified by precise statecharts, (ii) static checking the completeness and consistency of statechart specifications, (iii) generation of the C language representation of the control flow of the application, (iv) automated construction of test cases on the basis of structural test coverage criteria and (v) automatic construction of the source code of run-time verification procedures that aim at checking high-level safety properties. The application of these tools (as they are not certified) is intended to increase the confidence of the designers in the correctness of the design. This is assured by automating those systematic and tedious (typically error-prone) construction and verification procedures that were executed manually. The following parts of this short paper present the general concepts of formalization and tool development.

2 A Formal Operational Semantics for UML Statecharts

Automated checking and implementation of systems specified by statecharts needs to assign unambiguous meaning to statecharts. As the UML standard does not define an unambiguous operational semantics, multiple approaches have been published in the literature. Common drawbacks of these formalisms are that (i) they focus on quite restricted subsets of statechart artifacts, (ii) they were developed for old versions of UML and (iii) their model-checking point of view results in their extensive use of mathematical formalisms that are hard to understand for software engineers. We decided to define a formal semantics that is both mathematically well established and easily applicable in the engineering practice. The key steps of the approach are as follows: (i) first we establish the syntactic foundations by introducing the concept of precise statecharts and defining their metamodel; then (ii) we outline a formalism for explicit representation of compound transition and activity structures, finally (iii) we outline the definition of semantics for statecharts by a Kripke transition system and the translation of this formalism to easy to understand imperative algorithms.

The syntactic basis of our approach is the UML statechart metamodel. In order to rule the complexity, we considered junction and choice pseudostates, history vertices and facilities for embedding state machines as advanced constructs (shorthand notations that make the visual modeling more comfortable) and defined a set of formal transformation rules for their substitution with basic concepts [1]. From this point on, we will focus on those statecharts that contain basic constructs only (or are mapped to this form) and we will call these statecharts as precise statecharts (PSC).

From the point of view of software engineers, the main deficiencies of precise semantics definition are related to the transition structures (fork, join), and the ordering of activities when a compound transition is fired. We introduced the following concepts:
- Transition conglomerates: There are many cases when some transitions of a statechart can not be considered in isolation, e.g., input and output transitions of a forking pseudostate. In order to facilitate consistent and uniform discussion of these
compound transition structures we introduced the concept of transition conglomerates grouped into six classes (Fig. 1).

*Fig. 1. Transition Conglomerate Classes*

- **LCA and priority:** In the context of transition conglomerates we were able to provide a precise and intuitive formalization of the concepts of least common ancestor region (LCA), priority and conflict relations.
- **Compound activity structures:** When firing a transition conglomerate several activities are to be performed in a single step. The UML standard does not introduce a concept for handling compound activity structures either; however the unambiguous representation of strict subsequence relations and possibilities for parallel execution would be highly beneficial for exploiting the parallel processing capabilities of modern computing platforms. In order to overcome this weakness of the standard we introduced a formalism representing compound activity structures based on PERT graphs.

We introduced these concepts into the metamodel of precise statecharts and specified the formal operational semantics of UML 2.0 statecharts by a Kripke transition system (KTS). A KTS is defined by a three-tuple of states, state labeling function and the labeled state transition relation. In our case the states (statuses of the statechart) represent (i) the actual configuration of the statechart, (ii) the actual evaluation of variables and (iii) the actual phase of operation (e.g., run-to-completion step, terminated, etc.), thus the state labels represent three-tuples of this information. A transition of the KTS corresponds to a step between two statuses of the statechart representing (i) the event that triggered this step, (ii) the transition conglomerates that were fired in the step and (iii) the compound activity structure performed in the step.

We translated this declarative definition to easy to understand imperative algorithms implemented in the Microsoft AsmL executable specification language [1]. This way our approach is (i) mathematically well established (due to the rigorous formal semantics) and (ii) easily applicable in the engineering practice (due to the translation to imperative algorithms).

### 3 The Tool Chain

The AsmL imperative algorithms belonging to the formal semantics formed the basis of a statechart simulator tool. The modeler can construct an event sequence and the simulator calculates the transition conglomerates to be fired and the PERT graphs corresponding to the activity structures.

Besides this simulator, our tool set contains four other tools that are presented in the following subsections.
3.1 Static Checking of the Statechart Models

The compact representation of UML statecharts (including hierarchy, parallelism and nontrivial model elements) is a typical source of insufficiencies. Here we mention only the following three criteria [2]: (i) completeness – in order to prevent the state machine from dropping an event, in all possible statues of KTS, for all possible events, there must be a step transition defined which is triggered by the event; (ii) determinism – in each status, each event should trigger only a single step transition; and (iii) reachability – all states are reachable from the initial configuration.

These criteria can be formalized on the basis of the formal semantics, since it “unfolds” hierarchy, parallelism and the nontrivial model elements. Checking these criteria directly on the KTS requires the explicit generation of the KTS (i.e., the state space), which may lead to state space explosion in case of complex models. Accordingly, we adopted the approach of static checking: the criteria are adapted to syntactic terms (constructions of model elements) of precise statecharts to be able to check them directly on the model. We identified the hazardous scenarios belonging to the violation of the above criteria and on the basis of “reverse” semantic rules we defined static patterns that lead to these scenarios. The consistency and completeness checker tool applies a pattern matching algorithm to identify the concerned model elements.

3.2 Automatic Implementation of UML Statecharts

The implementation of a complex formalism like a statechart is definitely a nontrivial issue. The usual approaches (e.g., nested switch statements) are unable to handle such constructs as state refinement or parallel execution. Even the popular QHsm technique [3] is restricted to non-concurrent statecharts. The basis of our code generation is our metamodel of precise statecharts and our algorithms defining the operational semantics. We mapped the abstract concepts to the specialties of resource-constrained embedded systems: we substituted the complex AsmL algorithms (that calculate a possibly parallel execution order of various activities) with simple algorithms that calculate a single valid sequence of activities; substituted the recursive or mutually recursive function structures with iterative algorithms; introduced compact representation of configurations and similar data. We proved the semantic equivalence of these representations by comparing the corresponding algorithms line-by-line. In the final step, our tool implements the platform specific model in the ANSI-C language.

3.3 Automatic Test Generation for Statechart Implementations

To assess the quality of the test suites standards usually prescribe to meet certain coverage criteria, e.g., all statements and decisions must be at least once taken. Our test generator tool supports the model-based construction of a test suite satisfying model based coverage criteria (i.e., all states and transitions coverage) [5].

The components of our tool are depicted on Fig. 2. From the statechart model and a selected coverage criterion abstract test cases are generated that use the events described in the model. These abstract test cases are transformed to the format of the selected test execution engine. These concrete tests are then executed, and finally their code-based coverage is measured.
Fig. 2. Components of the Testing Environment

Our tool utilizes an external model checker to calculate the test cases: (i) the statechart is transformed into the input format of a model checker, (ii) each test requirement defined by the coverage criterion is formulated as a temporal logic expression, (iii) for each expression the negation of the formula is verified by the model checker. If there is an execution path in the model that does not satisfy the negated formula then it is presented by the model checker as a counter-example. This path becomes a test sequence belonging to the original test requirement. The input and output events are extracted from this path and saved as an abstract test case. The test transformation uses test skeletons that describe the test execution engine’s format (currently JUnit or Rational Robot) and templates (e.g., event dispatching and action verifying code).

3.4 Runtime Error Detection in Statechart Implementations

In case of safety critical systems random faults are typically addressed in run-time by various fault confinement or fault tolerance mechanisms based on efficient error detection. We present two runtime error detection techniques that aim at the detection of not only random operational faults but also statechart model refinement faults and implementation faults.

Model refinement faults are addressed by defining a temporal logic language for the specification of key dependability requirements in the context of early draft models and automatically checking that these temporal correctness criteria hold for the execution of the implementation. On the basis of the formal semantics we defined PSC-PLTL, the propositional linear temporal logic for precise statecharts. It includes (i) Boolean operators, (ii) temporal operators (the next-time X and until U, together with shorthand notations like the temporal future F and globally G) and (iii) atomic predicates of the language. The actual connection of PSC-PLTL and statecharts appear in the semantics of atomic predicates: they refer to the actual state configuration, the transition conglomerates fired, and the activities performed (this information resides in the state and transition labels of the KTS). For the runtime evaluation of PSC-PLTL formulae we elaborated an efficient method [6]. The source code of the runtime evaluation is generated automatically.

Implementation faults may originate from the misunderstanding of the model, usual programming bugs, or from the undesired interference of generated and manually inserted code. In our approach these faults are detected by a monitor that observes the runtime behavior of the implementation and compares it to the statechart model of the application. This approach was inspired by the idea of traditional watchdog processors (WP) [7] that observe the execution of a program and detect if it deviates from
the reference control flow specified by the control flow graph of the program. Although traditional watchdog solutions were successfully applied for detecting low-level errors, unfortunately none of them were capable of supporting such high-level reference structures as state refinement and concurrent execution featured by UML statecharts. In our approach we instrument the application in such a way that when taking a transition, it sends the labels of the source state, labels of the transition and labels of the target state to the monitor. These labels and the possible valid sequences of them are defined by the KTS, and the monitor is an automaton accepting the language formed by these valid sequences. This idea was implemented in a tool that generates the source code of the checker automatically on the basis of the PSC model.

4 Conclusions

In our paper we presented a coherent set of tools that support the UML statechart based design of event-driven applications that are typical in railway control systems.

The pilot application of our tools was a railway supervisory and control system [8]. Our experiments have shown that (i) for complex models, applications built according to our code generation approach delivered better performance with lower memory consumption than the common QHsm pattern, (ii) test suites for real-life models (2·10^8 states) can be generated, (iii) the PSC-PLTL checker is able to detect errors caused by model refinement faults, and (iv) the watchdog monitor detects most of the errors caused by implementation faults and a considerable number of errors caused by physical faults (as demonstrated by a software based fault injection campaign). The utilization of the tools and techniques is envisaged in the SAFEDMI (Safe Driver Machine Interface for ERTMS Automatic Train Control) European project.

References