Supporting Design and Development of Safety Critical Applications by Model Based Tools

Gergely Pintér, Zoltán Micskei and István Majzik

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

Abstract. Application of computer based systems in such safety critical areas like automotive on-board equipments, railway control systems etc. poses high dependability requirements against the corresponding software artifacts. This paper outlines a coherent tool-chain providing formally well-established support for the key phases of developing dependable software involving simulation, static model checking, automatic code generation, automatic test case synthesis and runtime error detection. Our approach focuses on modeling behavioral aspects of event-triggered state-based systems by using UML 2.0 statecharts as specification formalism. The application example analyzed in the paper was taken from the railway control domain.

Keywords: UML statecharts, static checking, assertions, test generation

1 Introduction

The software development standards for safety critical systems, like, e.g., EN 50128 for computerized railway control and protection systems, prescribe 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 effectively supported by automatic tools. However, the application of tools in the design and verification phases need 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.

1.1 Overview of the Tool-Chain

In the framework of a project supported by the Hungarian National Office of Research and Technology\footnote{Project nr. GVOP - 3.1.1 - 2004 - 05 - 0523/3.0} 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 statecharts called "Precise Statecharts". It includes all practically meaningful modeling concepts and has a fully defined operational semantics that was elaborated definitely for software engineers (instead of computer scientists). This formalization step allowed the development of the following support tools (Fig. 1): (i) simulation of the control flow specified by precise statecharts, (ii) static analysis of the completeness and consistency of statechart specifications, (iii) automatic synthesis of the C language representation of the application's control flow (i.e., code generation), (iv) automatic synthesis of test cases on the basis of structural test coverage criteria (i.e., test generation) 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 automatizing those systematic and tedious (typically error-prone) construction and verification procedures that were executed manually.

![Diagram of the Development Process](image)

Fig. 1 Overview of the Development Process

The conceptual structure of our work is as follows. As all the approaches outlined here are based on visual behavior models, we definitely need a solid understanding of the meaning of these models thus we define a formal operational semantics for statecharts of the UML 2.0 modeling language (Sec. 2). The rest of the discussion follows the scenario of a software development process. The statechart models prepared in the analysis and design phase are checked for consistency before stepping forward to the implementation phase thus Sec. 3 presents a static checking method for ensuring various consistency criteria in the context of statechart models. The next step of the development is the implementation of the models; Sec. 4 presents a code generation method for automatic source code level implementation of UML 2.0 statecharts. Since the effective testing of complex software is a challenging task, Sec. 5 outlines a test sequence generation method for automatic construction of trigger sequences that force a statechart implementation to traverse all the states or perform all the transitions. Finally Sec. 6 introduces two runtime error detection techniques for automatic identification of abnormal behavior performed by the operating software – these techniques are integrated to the software during the development chain.

### 1.2 Example Application

In order to illustrate our proposals, we will use a single example throughout the entire paper (Fig. 2). The example represents the over-simplified statechart model of a railway crossing controller being responsible for blinking the traffic lights on the road and moving the bar to a vertical (open) or horizontal (closed) position. Both top level states (grantedToCars and grantedToTrain) are decomposed to two concurrent regions each: the top regions operate the bar, the bottom ones are responsible for blinking the corre-
sponding lights (i.e., the two red bulbs in grantedToTrain and the single white bulb in grantedToCars). The operation of the two top-level states is similar, e.g., the top region of grantedToTrain contains two substates barMovingDown and barDown; the entry activity of barMovingDown starts the motor to move the bar down; having reached the horizontal state, a position switch sends the barBottomPS event moving the region to the barDown state (the exit activity of barMovingDown switches off the motor). The bottom region has two substates leftRedLightOn and rightRedLightOn, whose entry and exit activities switch the corresponding light bulb on or off; the transition between these states (i.e., the blinking) is triggered by a timer event. The crossing is equipped with two train sensors that send a trainApproaching or a trainGone event in case of the arrival and the passing of the train. These events trigger the transitions between the top-level states.

![Statechart of a Railway Crossing Controller](image)

Fig. 2 Statechart of a Railway Crossing Controller

2 A Formal Operational Semantics for UML Statecharts

Having decided to aim at automated checking and implementation of systems specified by statecharts, we are obviously in an essential need to assign unambiguous meaning to statecharts. Unfortunately the UML standard does not define an unambiguous operational semantics for statecharts thus multiple approaches have been published in the literature based on formal specification languages (e.g., PVS [2] or Abstract State Machines [3]), graph transformation [4], model transition systems [5] or by translating statecharts to Extended Hierarchical Automata and providing a semantics for these automata by a Kripke structure [6]. The previously published approaches were mainly targeted for model checking (e.g., converting UML statecharts to PROMELA, the input language of the SPIN model checker [6]). 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, seriously restricting this way their applicability in engineering practice. According to these considerations we decided to define a formal semantics for statecharts 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 meta-model; then (ii) we outline a formalism for explicit representation of compound transition and activity structures, finally (iii) we outline the definition of semantics for state-
charts by a Kripke transition system and the translation of this formalism to easy to understand imperative algorithms.

2.1 Solid Syntactic Foundations: The Metamodel of Precise Statecharts

The syntactic basis of our approach is the UML statechart metamodel. In order to rule the complexity we distinguished two sets of modeling facilities:

- **Basic concepts** are the ones that represent some fundamental artifacts of finite state-transition systems like state, transition, trigger etc.
- **Advanced concepts** are shorthand notations that make the visual modeling more comfortable but do not increase the expressive power of the language. We considered junction and choice pseudostates, history vertices and facilities for embedding state machines as advanced constructs 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 transformed to this form using the transformations mentioned above) and we will call these statecharts as precise statecharts (PSC). The metamodel of precise statecharts is shown in Fig. 3. The remaining differences between the PSC and the original UML 2.0 statechart metamodel are as follows: (i) the ambiguously defined “do activity” and “deferred trigger” concepts were removed and (ii) termination of execution is represented by “termination states” (newly introduced metaclass TerminationState derived from the metaclass State) instead of terminate pseudostates (this modification was needed for fixing the semantic inconsistency in the standard, i.e., representing a terminated status by a transient vertex).

![Fig. 3 Metamodel of Precise Statecharts](image-url)

2.2 Formalism for Compound Transition and Activity Structures

From the point of view of software engineers, the main deficiencies of precise semantics definition are related to the transition structures (fork, join etc.), and the ordering of activities when a compound transition is fired. To solve these problems, we introduced the following concepts:
- **Transition conglomerates**: It is easy to see that there are many cases when some transitions of a statechart cannot be considered in isolation, e.g., transitions connecting a fork pseudostate and target states (e.g., the ones originating in the fork pseudostate \( f_i \) and targeting \( \text{barMovingUp} \) and \( \text{whiteLightOn} \) in the example) are practically meaningless without the transition originating in a state and targeting the fork vertex (e.g., the one originating in \( \text{grantedToTrain} \) and targeting \( f_j \)). In order to facilitate consistent and uniform discussion of these compound transition structures (possibly involving multiple transitions and pseudostate vertices) we introduced the concept of transition conglomerates. An example for transition conglomerates is the structure mentioned above. We identified six transition conglomerate classes (Fig. 4) and formally defined them by tuples, e.g., the class mentioned in the example above can be described by the tuple \( (s_{\text{src}}, t, f, T_{\text{out}}, S_{\text{tgr}}) \) where \( f \) is the fork pseudostate, \( t \) is the transition targeting \( f \), \( s_{\text{src}} \) is the source state of \( t \), the set \( T_{\text{out}} \) contains the transitions originating in \( f \) and the set \( S_{\text{tgr}} \) contains the target states of the transitions in \( T_{\text{out}} \).

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Diagram" /></td>
<td><img src="image" alt="Diagram" /></td>
<td><img src="image" alt="Diagram" /></td>
<td><img src="image" alt="Diagram" /></td>
<td><img src="image" alt="Diagram" /></td>
<td><img src="image" alt="Diagram" /></td>
</tr>
</tbody>
</table>

**Fig. 4 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 [1].

- **Compound activity structures**: When firing a transition conglomerate multiple activities are to be performed: (i) first the exit activities of source states are executed in a bottom-up order (i.e., child states left before their parents); then (ii) effects of transitions are performed according to their sequence; finally (iii) the entry activities of target states are performed in a top-bottom order. Note that in parallel regions the activities belonging to the individual steps are performed in an unspecified order, even possibly in parallel. Unfortunately, similarly to compound transition structures, the UML standard does not introduce a high-level concept for handling compound activity structures either; however the uniform and 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 (Fig. 5 presents the PERT graph corresponding to the firing of the transition conglomerate of the previous example in the \( \text{barDown}, \text{rightRedLightOn}, \text{grantedToTrain} \) configuration).
2.3 Specification of the Semantics

Having formally defined the concepts related to compound transition and activity structures we introduced them into the metamodel of precise statecharts. Furthermore we defined a lightweight formalism for representing those variables that are referenced by guard predicates of transitions or various activities.

Finally, based on the toolkit outlined above we specified the formal operational semantics of UML 2.0 statecharts by a Kripke transition system (KTS). A KTS is defined by a three-tuple $K=\langle S, L, T \rangle$ over a $P_S$ set of state labels and $P_T$ set of transition labels, where $S$ is the set of states, $L: S \rightarrow P_S$ is the state labeling function and $T \subseteq (S \times P_T \times S)$ is the labeled state transition relation. In our case the state of the KTS will represent statuses of the statechart. The status concept was introduced by us representing (i) the actual configuration of the statechart, (ii) the actual evaluation of variables and (iii) the actual phase of operation (e.g., uninitialized, performing a run-to-completion step, terminated) thus the $p_S \in P_S$ 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 preformed in the step. The specification of the formal semantics is practically the definition of (i) the initial state of the KTS and (ii) the declarative definition of the T labeled transition relation.

Having specified the semantics by the KTS 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).

2.4 Simulation of Statechart Models

The AsmL imperative algorithms belonging to the formal semantics formed the basis of a statechart simulator tool (Fig. 6) in which the modeler can construct an event sequence (top part of the window) and let the application simulate the response of the state machine to this event sequence (here written in the text box at the bottom of the window). The simulator calculates the trajectory in the state space (on the basis of the firing transition conglomerates) and the PERT graphs corresponding to the activity structures. Moreover, it enables the interactive evaluation of transition guards if needed. In the example of the figure the railway crossing controller is provided the $trainGone$, $timer$,
timer, barTopPS, timer event sequence; the simulator calculates the five corresponding steps of the application and prints the resulting configuration (barUp, whiteLightOff, grantedToCars) as highlighted in the text box.

![Simulation of a Statechart Model](image)

**Fig. 6 Simulation of a Statechart Model**

### 3 Static Checking of the Statechart Models

The detection of insufficiencies like incompleteness and inconsistency of the statechart specification is crucial since during the verification steps the implementation is checked with regard to this specification, thus the insufficiencies will find their way to the final product. Not surprisingly, experience shows that the majority of computer-related accidents originate in specification errors [19]. The compact representation of UML statecharts (including hierarchy, parallelism and nontrivial model elements) is a typical source of such errors.

The most important completeness and consistency criteria can be formalized on the basis of the formal operational semantics of the statecharts, since it “unfolds” hierarchy, parallelism and the nontrivial model elements. Here we mention only the following three criteria [20]: (i) completeness – in order to prevent the state machine from dropping an event, in all possible statuses of KTS, for all possible events, there must be a step transition (in a special case an internal 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 – normally all states of a statechart are reachable from the initial configuration through a sequence of transitions.

Checking these criteria directly on the KTS requires the explicit generation of the KTS (i.e., the state space of the application), which may lead to the infamous problem of
state space explosion in case of complex models. Accordingly, we adopted the approach of static checking: criteria are adapted to syntactic terms (specific constructions of model elements) of precise statecharts in order to be able to check them directly on the model, without the need of generating the KTS. The concept of static checking is summarized in the following two steps:

- We use the rules of semantics in “reverse direction” to identify scenarios, i.e., state configurations and firing of transition conglomerates that are affected when a specific criterion is satisfied (or not satisfied). In this way hierarchy, concurrency, and priority scheme are taken into account.

- Since a criterion interpreted on a KTS may cover several scenarios in the UML statechart, the criteria can be decomposed into a set of sub-criteria. These sub-criteria are defined in syntactic terms of precise statecharts. For example, we can say that if a given basic state is not in a concurrent region, then the set of transitions that may fire is composed by the set of transitions of this basic state and the transitions of its parents. If we virtually inherit the transitions of parents to the basic state, then the completeness criterion can be formulated as follows: For each basic state, for all possible events, there must be a transition defined, and no checking is necessary at the composite (parent) states.

In our previous work [15] we mapped the criteria to the model elements of UML 1.3 statecharts, thus in this case our task was reduced to the adaptation of this mapping to precise statecharts. The advantage of the PSC formalization turned out clearly: the existence of the PSC metamodel simplified this task and the corresponding well-formedness rules even filtered out some cases of incomplete specification.

Our consistency and completeness checker tool provides a way to specify those regions of the statechart in which static checking is required (e.g., critical operating modes), implements the static checking rules, and generates the list of model elements in which one of the criteria is violated. Note that the tautology and overlapping of the truth of Boolean guard conditions is also checked statically. The screenshot of our static checker tool (Fig. 7) presents an aspect of investigating the consistency of the railway crossing controller’s statechart: we selected the grantedToTrain superstate and checked that all states of the statechart are statically reachable.

![Checker for Precise Statecharts](image)

**Fig. 7 The Static Checker Tool**
4 Automatic Implementation of UML Statecharts

The automatic implementation of such a complex formalism as a statechart is definitely a nontrivial issue. The usual naïve approaches (e.g., implementing the state-transition logic by nested switch statements or state-transition tables) are unable to handle such fundamental constructs as state refinement or parallel execution, not even the well-known State design pattern [8] is capable of supporting these concepts. The solutions published in various research papers are unfortunately also restricted to a subset of UML statechart features. Even the best-known Quantum Hierarchical state machine (QHsm) implementation technique explicitly proposed for embedded systems by Samek [9, 10] is restricted to non-concurrent statecharts. Having taken into consideration the lack a of full-featured embeddable solution we decided to adapt the Model Driven Architecture (MDA) [7] initiative of the Object Management Group (OMG) for this challenge. The MDA process consists of three phases: (i) platform-independent modeling, (ii) platform-specific modeling and (iii) implementation; in the usual illustration of MDA (Fig. 8) we distinguish the metamodel level (corresponding to modeling languages used in various steps) and the model level (corresponding to the actual models built in the appropriate language). Below we outline our case study, the implementation of statecharts on resource-constrained embedded platforms.

In the platform-independent modeling (PIM) phase the system is modeled barely focusing on the services to be delivered, data structures to be implemented etc. without taking into consideration any peculiarities of the target platform. In our case the PIM phase corresponds to the metamodel of precise statecharts and our algorithms defining the operational semantics.

![Fig. 8 Overview of the Model Driven Architecture](image)

The goal of the platform-specific modeling (PSM) phase is to map the abstract concepts of the PIM phase to the specialties of the target platform still remaining at the abstract modeling level (i.e., using stereotyped UML diagrams). In our case we had to (i) first identify those characteristics of resource-constrained embedded systems that may require some modifications of the platform-independent semantics and (ii) actually carry out the necessary modifications both in the metamodel and the algorithms. The dominant characteristics of resource-constrained embedded systems identified by us were as follows: (i) low computing power, (ii) serious memory constraints, (iii) lack of hardware support for parallel execution and (iv) need for deterministic or even real-time operation. In correspondence to these observations we carried out modifications in the metamodel and the algorithms specifying the operational semantics as follows:
we substituted the complex algorithms (that calculate a possibly parallel execution order of various activities) with simple algorithms that calculate a single valid sequence of activities (reducing this way the processing power requirements and taking into consideration the lack of parallel execution possibilities);

- substituted the recursive or mutually recursive function structures with iterative algorithms (supporting this way the pre-calculation of execution times for real-time operation);

- introduced compact representation of configurations and similar data (reducing this way the memory consumption).

The resulting platform-specific language consisted of a modified metamodel of precise statecharts and a set of modified algorithms. We proved the semantic equivalence of the PIM and PSM representations by comparing the corresponding algorithms line-by-line and discussing the modifications and their correctness.

The final step of MDA is the implementation phase. The goal of this step is to implement the platform specific model in a programming language that seamlessly fits the target platform. In our case this means the implementation of the PSM metamodel and algorithms in the ANSI-C language, as data structures and functions, respectively. In order to achieve this, first we prepared an annotated metamodel of precise statecharts indicating how to implement the corresponding model element in C (e.g., by a built-in data type, a structure, or an enumerated type) then implemented the data structures and the algorithms. We proved the correctness of the implementation similarly to the PIM-PSM step.

Having built the theory behind the PIM, PSM and implementation phases we implemented the process in an automatic code generator. Our implementation expects the models in the XML metadata interchange (XMI) format supported by most of the UML 2.0 modeling tools, enabling this way the seamless integration into popular environments. Our experiments have shown that (besides supporting the implementation of all model elements) for complex models (deep state hierarchies, large number of states) applications built according to our approach delivered better performance with lower memory consumption than the ones corresponding to the QHsm pattern.

5 Automatic Test Generation for Statechart Implementations

Testing is the most commonly used verification method in software development. However, manual testing could be very time consuming and usually needs expert knowledge. 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 construction of a test suite satisfying model based coverage criteria (i.e., all states and transitions coverage) [12].

The high-level components of our tool are depicted on Fig. 9. From the statechart model and a selected coverage criterion test cases are generated. These test cases use the events described in the model, hence we refer to them as abstract test cases. When the statechart is implemented (either manually or by the code generator described earlier), the model elements have to be mapped to program structures. The running of the tests is automated using a test execution engine, thus the abstract test cases need to be trans-
formed to the format of the selected execution engine. These concrete tests are then executed, and finally their code-based coverage is measured.

Our tool utilizes an external model checker to calculate the test cases. The following steps are performed during the test generation: (i) the statechart is transformed into the input format of the SPIN 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 that satisfies the original test requirement. Finally, (iv) the input and output events that form the executable test sequence are extracted from the counter-example 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 model specific templates. The transformation task consists of the following steps: (i) the event names have to be mapped to their representation in the code, (ii) the setup and clean up code for the test suite has to be written, and (iii) the templates containing the implementation specific event dispatching and action verifying code have to be created. At the final step (iv) the transformation fills the test skeleton with the sequences in the test cases.

In the following the above steps are illustrated on the railway crossing example. The tester selects the all states coverage criterion. This specifies the requirements that during the execution of the test suite all states has to be entered. For each of the requirements, i.e., for each state, a linear temporal logic (LTL) formula is generated asserting that the state is not reached. These formulae are checked in SPIN, and if the state can be reached from the initial configuration then the formula is violated and a counter-example is generated, e.g., in the case of the whiteLightOff state the following sequence is generated as a counter-example: trainGone, {switchWLOn, startMotorUp}, timer, switchWLOff.

The default configuration of the model checker is optimized for an exhaustive search of the full state space. However, in test generation the goal is to find a counter-example visiting only as few states as possible. Thus, a specific configuration of the model checker is needed for efficient test generation. Through performing several experiments to measure the effect of the different options offered by the SPIN tool, the following parameter set was selected that is suitable for generating minimal length test sequences. The depth of the depth-first search algorithm is limited and the size of the hash table used for storing the states internally is set to reflect the total number of states in the statechart. Breadth-first search turned out to be too slow, although it found the minimal length tests. For larger models, SPIN’s bit-state hashing state compression technique was turned on to handle state space explosion. Each test case covers multiple test requirements, thus the test generator searches a new test case for a requirement only if it is not already covered. With these settings test suites for real-life models were successfully generated (a synchronization protocol modeled by 31 statechart states and 174 transitions, with a state space consisting of $2 \cdot 10^9$ states).
6 Runtime Error Detection in Statechart Implementations

Even after a carefully designed and performed testing process there may some faults residing in the system that is deployed to the target platform. In case of safety critical systems these issues are typically addressed in run-time by various fault containment or fault tolerance mechanisms. The entry point of these mechanisms is the actual detection of an erroneous situation. This section presents two runtime error detection techniques aiming at the detection of (i) model refinement faults and (ii) implementation and operational faults. Model refinement faults occur during the model elaboration process if a refined statechart model violates some dependability constraints defined in the context of an earlier (draft) model. In our approach these 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. Implementation faults may originate from the misunderstanding of the model or usual programming bugs in case of manual coding, or from the undesired interference of generated and manually inserted code in case of automatic code generation. In our approach these faults are addressed by a tool that observes the runtime behavior of the implementation and compares it to the statechart model of the application.

6.1 Detection of Errors Caused by Model Refinement Faults

Temporal logic (TL) languages were originally suggested by Pnueli [15] for reasoning about concurrent programs. The core concept of checking linear temporal criteria is to define a finite state-transition system representing an abstraction of the application and check that specific propositions hold for execution traces of the application. This abstraction is usually presented by a Kripke-structure (KS). There are relatively few proposals for mapping TL languages to artifacts of UML statecharts. The FNLOG language [11] primarily aimed at model checking, TLCharts [14] are extensions of Harel statecharts. Having taken into consideration the lack of a temporal logic language explicitly designed for runtime error detection in statechart implementations, we decided to construct such a language based on our statechart semantics.

Since the semantics was defined by a Kripke transition system, its translation to a KS was barely a syntactic rewriting (i.e., labels assigned to transitions are copied to the labels of the corresponding target state). Having translated the KTS to a KS, the definition of the temporal logic language consists of the specification of the (i) Boolean operators, (ii) temporal operators and (iii) atomic predicates of the language. We used the usual Boolean operators and the next-time (X) and until (U) temporal operators. The informal meaning of temporal operators is as follows: (i) $Xp$ is true if for the next state of the KTS $p$ holds; (ii) $pUq$ is true if sometime in the future for a state of the KTS $q$ will hold and until that point $p$ holds for every states. We introduced shorthand notations like the temporal future (F) and globally (G) operators ($Fp$ is true if for a state of the KTS eventually $p$ holds, $Gp$ is true if during the entire operation of the system $p$ holds for all states). The actual connection of the language and statecharts appear in the semantics of atomic predicates: we defined various predicates for referring to the actual state configuration of the statechart, the transition conglomerates fired, and the activities performed; note that the information we are referring to resides in the state and transition labels of
the KTS. We call this language PSC-PLTL (propositional linear temporal logic for precise statecharts).

PSC-PLTL enables the definition of safety and liveness criteria. In the context of the railway crossing controller example, e.g., “whenever a train is approaching, one of the red lights should be on and finally the bar should be down” can be expressed as $G(\text{TrainApproaching} \rightarrow ((S_{\text{LeftRedLightOn}} \lor S_{\text{RightRedLightOn}}) \land F(S_{\text{BarDown}})))$. In this example we used the atomic predicate syntax $S_x$, indicating that state $x$ is active and $E_y$ indicating that the most recent transition was triggered by the event $y$. For the runtime evaluation of PSC-PLTL formulae we elaborated an efficient method [15]. The source code of the runtime evaluation (specific for the criteria) is generated automatically.

### 6.2 Detection of Errors Caused by Implementation Faults

Our proposal for detecting implementation faults was inspired by the idea of traditional watchdog processors (WP) [16]. A WP is a relatively simple co-processor that observes the execution of a program on the main CPU and detects if the actual execution of the program deviates from the reference control flow specified by the control flow graph (CFG) of the program. Nodes of the CFG are subsequent branch-free blocks of instructions and directed edges of the graph correspond to syntactically allowed branches. The goal of the WP is to check whether the actual execution is a valid path in the CFG. Although traditional watchdog solutions were successfully applied for detecting low-level behavior errors, unfortunately none of them are capable of supporting such high-level reference structures as state refinement and concurrent execution featured by UML statecharts.

Our proposal was inspired by the idea of watchdog program solutions [17, 18] and directly based on the KTS defining the semantics of the statechart. We can imagine the application specified by the statechart, i.e. the KTS, as an automaton that when taking a transition $(S_{\text{SRC}} \times P_t \times S_{\text{TARG}}) \in T$ (labeled transition, actually sends the $(L(S_{\text{SRC}}), P_t, L(S_{\text{TARG}}))$ tuple to its output (i.e., labels of the source state, labels of the transition and labels of the target state). These output tuples can be considered as words of a language thus all possible valid executions of the KTS define all possible valid sentences of the language. From this point of view our task was to define a checker for this language (i.e., an automaton accepting the language). Since the language was defined by a finite state machine (KTS) the checker can also be implemented by a simple finite state machine by formally deriving its transition relations from the statechart semantics. This idea was implemented in a tool that generates the source code of the checker (as a software module called PSC-WD) automatically on the basis of the PSC model.

The error detection potential of the two approaches was demonstrated by a fault injection campaign involving (i) model refinement faults, (ii) implementation faults and (iii) physical faults simulated by a SWIFI tool. Our experiments have justified that the PSC-PLTL checker detected most of errors caused by model refinement faults and the PSC-WD detected most of errors caused by implementation faults and a considerable number of errors caused by physical faults.
7 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 safety critical systems. Instead of describing the detailed algorithms (that fit into long research reports) we concentrated on the concepts that are reusable both by designers and statechart-based tool developers, i.e., how the semantics can be formalized using an engineering language, and how the formal semantics can be a basis of tools that implement several methods prescribed by the safety standards.

The pilot application area of our tools was a railway supervisory and control system that is connected to the interlocking system of a railway station either locally or through remote interfaces [21]. We developed the test cases of the critical synchronization protocol, and constructed the run-time PSC-WD module and the checker code corresponding to various safety criteria formalized in PSC-LTL. The application of our tools showed that model based design can be cost efficient if the model is not only a documentation of design decisions but can be used as a basis to perform automated checking and implementation steps that replace tedious manual work.

The limitations of our tools appear first of all in the test generation phase as state space explosion may occur. The scalability of the other tools with respect to the size of the model is not a critical issue due to the moderate complexity of source code generation techniques and the adopted static checking technique in the consistency verification step.

References

8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns Elements of Reusable Object-Oriented Software. Addison Wesley, 1994.