Annales Univ. Sci. Budapest., Sect. Comp. 30 (2009) 61-78

SUPPORTING DESIGN AND DEVELOPMENT OF SAFETY CRITICAL APPLICATIONS BY MODEL BASED TOOLS

G. Pintér, Z. Micskei and I. Majzik

(Budapest, Hungary)

Abstract. Application of computer based systems in safety critical areas like automotive on-board equipments, railway control etc. poses high dependability requirements against 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, test case synthesis and runtime error detection. Our approach focuses on modeling behavioral aspects of event triggered state-based systems using UML 2.0 statecharts as specification formalism. The application example analyzed in the paper was taken from the railway control domain.

1. Introduction

The software development standards for safety critical systems, like, e.g., EN 50128 for computerized railway control and protection systems, prescribe several mandatory or highly recommended methods and techniques to reach a given safety integrity level (SIL), e.g. static analysis, failure assertion programming and structural testing. Some of these methods can be effectively supported by automatic tools, however, their application in the design and verification phases need (i) clear understanding of the corresponding formalisms and models and (ii) the interpretation of delivered results. 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 and usability problems appeared in connection with the formal semantics developed so far. In order to overcome these issues we defined a novel formal operational semantics of statecharts definitely for being used by *software engineers*.

On the solid basis of the formal semantics we constructed a coherent chain of tools to increase the confidence of the developers in the correctness of their design by substituting tedious and typically error-prone manual work with automatic solutions (Figure 1): (i) a *simulator* for statecharts, (ii) *static analysis tools* for checking of the completeness and consistency of statechart specifications, (iii) an *automatic code generator* for synthetizing source code on the basis of statecharts, (iv) a *test generator* for achieving various structural test coverage criteria and (v) a synthesis method for the automatic construction of *run-time verification procedures* that aim at checking high-level safety properties.

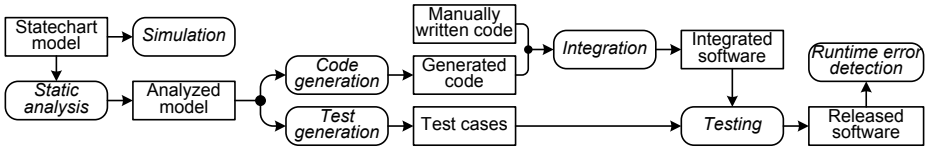

Figure 1. Overview of the development process

The conceptual structure of our work is as follows. Section 2 introduces our formal semantics for statecharts; the rest of the discussion follows the scenario of a software development process: statecharts prepared in the analysis and design phases are *checked for consistency* by static checking tools (Section 3) then *implemented* by the supported of code generators (Section 4); since the effective testing of complex software is a challenging task, Section 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. Section 6 introduces two runtime *error detection techniques* for automatic identification of abnormal behavior performed by the running software. Our experiences gained during the application of the toolchain are summarized in Section 7, finally Section 8 concludes the paper and outlines the directions of future research.

In order to illustrate our proposals, we will use a single example throughout the entire paper (Figure 2). The example represents the simplified statechart

[1]Project number: GVOP-3.1.1-2004-05-0523/3.0

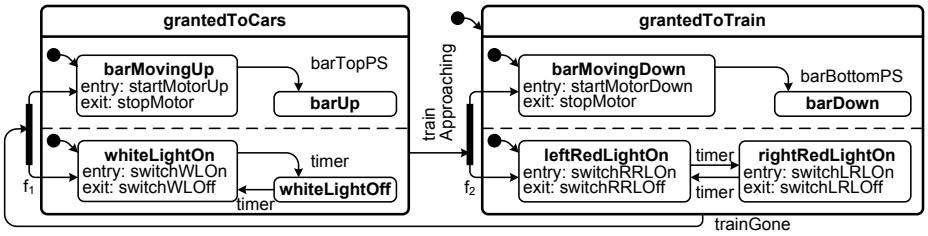

Figure 2. Statechart of a railway crossing controller

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 corresponding 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.

2. A formal operational semantics for 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 a formal semantics for statecharts thus multiple approaches have been published in the literature based on formal specification languages [1], graph transformation, model transition systems [2] or Extended Hierarchical Automata [3]. Common drawbacks of most previously published formalisms are that (i) they focus on quite *restricted subsets* of statechart artifacts, (ii) they were developed for *obsoleted 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. 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.

2.1. Syntactic foundations: The metamodel of precise statecharts

The syntactic basis of our approach is the UML metamodel. In order to rule the complexity we distinguished two sets of modeling facilities: (i) *basic concepts* are the ones that represent some fundamental artifacts of finite state-transition systems like state, transition, trigger etc. while (ii) *advanced concepts* (junction, choice, history and submachines) that are shorthand notations making the visual modeling comfortable but do not increase the expressive power of the language. In our approach we focus on statecharts containing basic concepts, only all call such statecharts *precise statecharts* (PSC). In order to maintain the support for convenient advanced concepts we formally define a set of formal transformation rules for substituting advanced constructs with basic ones [4] thus our achievements support this way all relevant UML statechart modeling artifacts.

2.2. Formalism for compound transition and activity structures

From the point of view of software engineers, the main deficiencies of the standard are related to compound transition structures, and the ordering of activities when a compound transition is fired. To solve these problems, we introduced the *transition conglomerate* and *compound activity structure* concepts.

It is easy to see that there are many cases when some transitions of a statechart can not be considered in isolation, e.g. transitions connecting a fork pseudostate and target states (e.g. the ones originating in the fork pseudostate f_1 and targeting *barMovingUp* and *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 *grantedToTrain* and targeting f_1). 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 (Figure 3) and formally defined them by tuples, e.g. the class mentioned in the example above can be described by the tuple $(s_{\mathrm{src}}, t, f, T_{\mathrm{out}}, S_{\mathrm{trg}})$ where f is the fork pseudostate, t is the transition targeting f, s_{src} is the source state of t, the set T_{out} contains the transitions originating in f and the set S_{trg} contains the target states of the transitions in T_{out}. 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 [4].

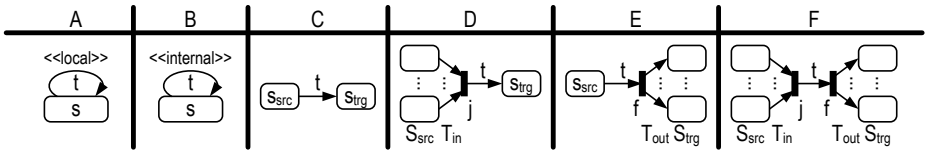

Figure 3. Transition conglomerate classes

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 their unambiguous representation would be highly beneficial for exploiting the parallel processing capabilities of modem computing platforms. In order to overcome this weakness of the standard we introduced a formalism representing *compound activity structures* based on PERT graphs (Figure 4 presents the PERT graph corresponding to the firing of the transition conglomerate of the previous example in the *barDown*, *rightRedLightOn*, *grantedToTrain* configuration).

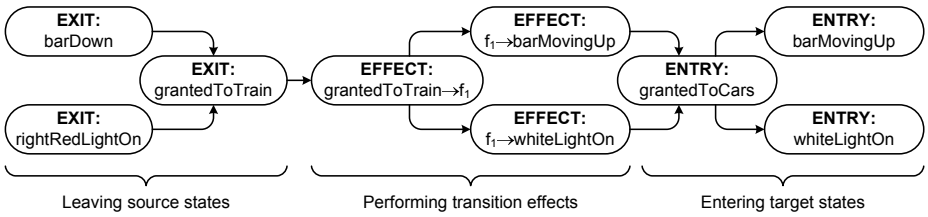

Figure 4. Activities performed on firing a transition conglomerate

2.3. Specification of the semantics

Based on the toolkit outlined above we specified the formal semantics by a Kripke transition system (KTS). A KTS is defined by a three-tuple $K = (S, L, T)$ over a P_S set of state labels and P_T set of transition labels, where S is the set of states, $L : S \to P_S$ 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 represents *statuses* of the statechart involving (i) the actual *configuration*, (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 and (iii) the *compound activity structure* performed 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 it to *imperative algorithms* implemented in the Microsoft AsmL executable specification language [5].

To put together the discussion above the key beneficial features of our semantics as compared to previously published ones are as follows: (i) it is *directly based* on the *most recent UML metamodel* thus the models exported from modeling tools are directly usable without any intermediate formats; (ii) both *mathematically well-established* and *easily usable* in the engineering practice; (iii) provides an unambiguous formalism for the representation of *compound transition structures*; (iv) provides a PERT-graph based formalism for the representation of *compound activity structures* enabling this way the exploitation of parallel computing resources of modern platforms and (v) the *executable definition of the semantics* enable *straightforward implementation* in imperative programming languages and *proving the correctness* of the implementation.

The first element of our toolchain the *statechart simulator tool* (Figure 5, left) directly implements the semantics: the modeler constructs an event sequence and lets the application simulate the response of the state machine. The simulator calculates the trajectory in the state space and the activity structures to be performed. In the example of the figure after receiving the *trainGone*, *timer*, *timer*, *barTopPS*, *timer* event sequence the railway crossing reaches the (*barUp*, *whiteLightOff*, *grantedToCars*) configuration.

3. Static checking of statechart models

The detection of insufficiencies like *incompleteness* and *inconsistency* of the statechart specification is crucial since during the verification steps the imple-

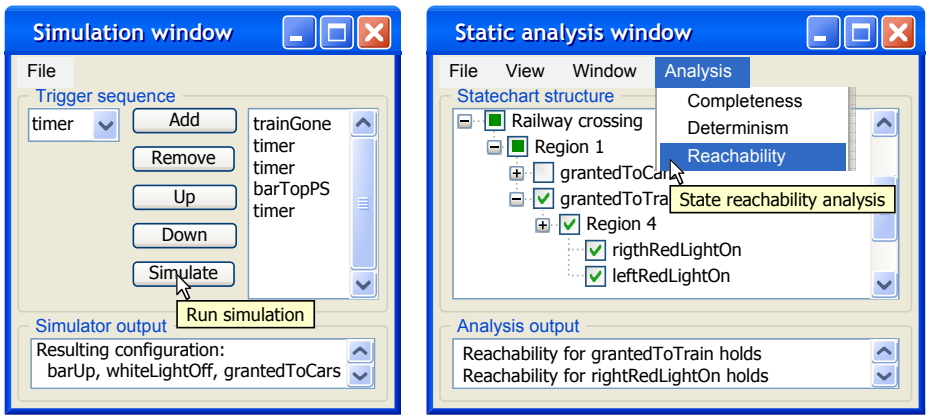

Figure 5. Simulation and analysis of statechart models

mentation 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* [6]. 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 [7] (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 (possibly an internal transition) 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.

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 statechart, the criteria can be decomposed into a set of sub-criteria. These sub-criteria are defined in syntactic terms of PSCs, e.g. 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 [8] we mapped the criteria to UML 1.3 statecharts, thus in this case our task was just 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 incomplete specifications.

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. Our static checker tool in right part of Figure 5 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.

4. Automatic implementation of statecharts

The *automatic implementation* of such a complex formalism as a statechart is definitely a nontrivial issue. The usual naive 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 [9] 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 [10, 11] is restricted to non-concurrent statecharts. Having taken into consideration the lack of a full-featured embeddable solution we decided to adapt OMG's Model Driven Architecture (MDA) [12] initiative 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 (Figure 6) we distinguish the metamodel level (corresponding to modeling languages used in

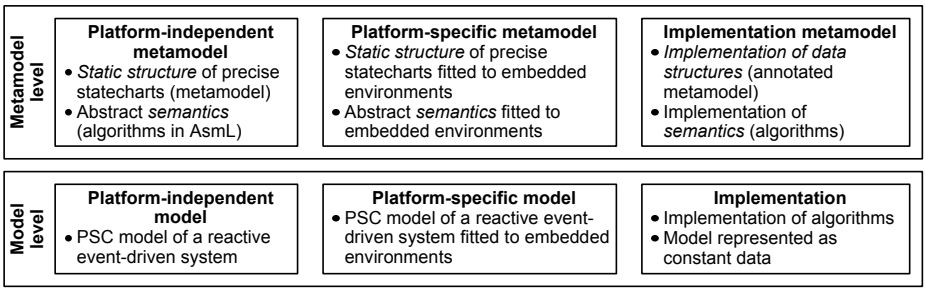

Figure 6. Overview of the model driven architecture

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.

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 PIM 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 the observations above we carried out modifications in the metamodel and the algorithms specifying the operational semantics. We substituted the complex algorithms that calculate a possibly parallel execution order of various activities with more 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. Recursive or mutually recursive function structures were substituted with *iterative algorithms* (supporting this way the pre-calculation of execution times for real-time operation). Finally configurations and similar data structures were organized into a compact representation (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].

5. Automatic test generation for statecharts

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 taken at least once. Our *test generator tool* supports the construction of a test suite satisfying *model based coverage criteria* (i.e. state and transition coverage) [13].

The high-level components of our tool are depicted on Figure 7. 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 transformed to the format of the selected execution engine. These concrete tests are then executed, and finally their code-based coverage is measured.

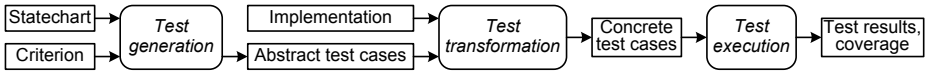

Figure 7. Components of the test generator

Our tool utilizes an external model checker for calculating test cases. The following steps are performed during 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 test cases.

Below we illustrate the steps outlined above for meeting the *all state coverage* criterion in the context of the railway crossing example. In order to enter all states by the test suite 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*.

SPIN's default configuration is optimized for an exhaustive search of the full state space. However, in test generation the goal is to find a counter-example visiting 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.

6. Runtime error detection

Even after a carefully designed and performed testing process there may some faults reside in the delivered system. In case of safety critical environments these issues are typically addressed by various runtime fault tolerance mechanisms whose entry point 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 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 automatically generated and manually written code. 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 for reasoning about concurrent programs [14]. The core concept of checking 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). As previously published statechart-related TL languages focus on model checking [15] or use Harel statecharts as reference [16] (i.e. not the UML dialect) we decided to construct a TL language based on our semantics explicitly focusing on runtime error detection in UML statechart implementations.

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 ($\mathcal{X}$) and until ($\mathcal{U}$) temporal operators. The informal meaning of temporal operators is as follows: (i) $\mathcal{X}p$ is true if for the next state of the KTS p holds; (ii) $p\mathcal{U}q$ 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 ($\mathcal{F}$) and globally ($\mathcal{G}$) operators ($\mathcal{F}p$ is true if for a state of the KTS eventually p holds, $\mathcal{G}p$ 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 $\mathcal{G}(E_{\text{trainApproaching}} \rightarrow ((S_{\text{leftRedLightOn}} \vee S_{\text{rightRedLightOn}}) \wedge \mathcal{F}(S_{\text{barDown}})))$. 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 [17]. 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) [18]. A WP is a 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 [19, 20] 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 $t = (s_{\mathrm{SRC}} \times P_T \times s_{\mathrm{TRG}}) \in T$ (labeled) transition, actually sends the $(L(s_{\mathrm{SRC}}), P_T, L(s_{\mathrm{TRG}}))$ tuple to its output (i.e. labels of the source state, transition and 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 (i.e. the KTS) the checker can also be implemented by a 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.

7. Application experiences

According to experiments and estimations published in the literature the application of model-based development techniques promises a significant improvement of software quality and reduction of development costs. In case of the traditional development of a safety critical (SIL-2) railway remote control system the number of faults found was 11 faults/kSLOC (11 faults in thousand lines of source code); according to precise estimations and pilot experiments [21] the number of faults in the same product would be reduced to (ii) 6 faults/kSLOC in case of UML-based requirement analysis and design with manual implementation, (iii) 2.5 faults/kSLOC in case of UML-based requirement analysis, modeling and MDA-based automatic implementation and (iv) 0.25 faults/kSLOC by the support of formal analysis tools. In similar estimations the development time can be reduced to 42% of the traditional approach (i.e. textual requirement analysis and design with manual implementation) by UML-based analysis and design, to 11% by MDA-based automatic implementation. Obviously the numbers cited above are based on considerable amount of optimistic estimation; according to our knowledge there were no such detailed experimental analyses published in the literature – note that this kind of analysis and comparison is extremely labour intensive and expensive thus no companies are interested in publishing data about their development efficiency. Obviously neither can we present here end-to-end comparisons just enumerate some promising experiences with our toolchain below.

We successfully *generated the source* of various event-driven pilot applications and deployed them to a wide range of computing environments ranging from powerful PCs to embedded devices equipped with 8-bit microcontrollers and re-

stricted memory [22, 5]. The execution of applications was *simulated* before code generation to visually demonstrate their behavior. In the future we plan to experiment with the exploitation of parallel computing resources of enterprise-level platforms by generated code.

The *static checking tool* was used in a proof-of-concept experiment for the analysis of the top-level statechart of a safety critical SIL-2 railway control application (freedom from deadlocks and determinism was proven).

The effectiveness of our two runtime *error detection techniques* was demonstrated by a *fault injection campaign* involving both model refinement, implementation and physical faults [5]. Our experiments have justified that the PSC-PLTL checker detected most of errors caused by model refinement faults while the PSC-WD detected most of implementation-related errors and a considerable number of physical issues. The key idea of PSC-WD will be used for runtime detection of control flow errors in the top-level control structure of a safe train driver-machine interface developed in the SafeDMI EU project [23]. The PSC-PLTL checker was used for the analysis of log files produced by another railway control application.

Upon the first application of our *test generation* technique to a real life industrial example the traditional problem of model checkers, namely state space explosion was encountered. The case study was a protocol that synchronizes status and control bits between two computers in a distributed control system in presence of anticipated faults. The model consisted of 5 objects with event queues, 31 states and 174 transitions. In the generated PROMELA code the state vector (which identifies a state) was 216 bytes long, and during the exhaustive verification more than 2×10^8 states were explored. Thus, the complete verification would have needed approximately 40GB of memory; however using bit-state hashing and proper parameterization as described in Sect. 5, test suite generation for the all transition coverage criterion was feasible on a machine with 1GB of RAM in 65 minutes.

8. Conclusions

This paper has presented a coherent set of tools supporting the design and development of safety critical, state based event driven systems. Key contributions of our work involve (i) the introduction of an unambiguous formal operational semantics for UML statecharts, (ii) a simulator facility, (iii) a static model checker tool, (iv) automatic code and (v) test generation approaches and (vi) two runtime error detection methods. The discussion focused on outlining reusable ideas of our approaches while involved details of the actual implementation are presented in corresponding research reports [4, 5].

Some key novelties and improvements of this approach as compared to our previous proposals and other solutions published previously in the literature are as follows: all facilities are built on the *same solid formal foundation*; the code generation solution supports practically the entire UML statechart modeling toolkit; the test generation facility scales well to practical problems and our runtime error detection facilities are explicitly targeted to UML statecharts. Key limitations to be mentioned originate from theoretic reasons: although the test generation method appears to behave well in case of practical models, the issue of possible state space explosion can not be totally eliminated.

The viability of our tools were demonstrated in some pilot projects, e.g. in case of a railway supervisory and control system that is connected to the interlocking system of a railway station [24]. We developed the test cases of the critical synchronization protocol and constructed the run-time error detection code corresponding to various safety criteria formalized in PSC-PLTL. The effectiveness of the code generation solution was demonstrated by automatically synthesizing source code for a microcontroller-based embedded platform. Pilot experiments have shown again that ever more popular model based approaches promise considerable increase in productivity and reduction of development costs by replacing tedious manual work with automatic tools.

References

[1] **Börger E., Cavarra A. and Riccobene E.,** Modeling the dynamics of UML state machines, *Proceedings of the International Workshop on Abstract State Machines, Theory and Applications,* LNCS **1912**, Springer Verlag, 2000, 223-241.

[2] **Varró D.,** A formal semantics of UML statecharts by model transition systems, *Proceedings of the 1st International Conference on Graph Transformation (ICGT-2002)*, eds. A. Corradini, H. Ehrig, H.J. Kreowski and G. Rozenberg, LNCS **2505**, Springer Verlag, 2002, 378-392.

[3] **Latella D., Majzik I. and Massink M.,** Towards a formal operational semantics of UML statechart diagrams, *FMOODS'99, The Third IFIP International Conference on Formal Methods for Open Object-based Distributed Systems, Firenze, Italy, February 1999*, 331-347.

[4] **Pintér G. and Majzik I.,** *Formal operational semantics for UML 2.0 statecharts*, technical report, Budapest University of Technology and Economics,

2005. (Prepared for the Hungarian Academy of Sciences (MTA/TKI) in the framework of the Embedded Systems Research Group at BUTE/DMIS.)

[5] **Pintér G.,** *Model based program synthesis and runtime error detection for dependable embedded systems*, PhD thesis, Budapest University of Technology and Economics, Department of Measurement and Information Systems, Budapest, Hungary, 2007.

[6] **Leveson N.G.,** *Safeware: System safety and computers,* ACM Press, 1995.

[7] **Pap Z., Majzik I., Pataricza A. and Szegi A.,** Completeness and consistency analysis of UML statechart specifications, *Proceedings of the 4th IEEE International Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS- 2001), Győr, Hungary, April 2001*, 83-90.

[8] **Pintér G. and Majzik I.,** Run-time verification of statechart implementations, *Architecting Dependable Systems,* eds. C. Gacek, A. Romanovsky and R. de Lemos, LNCS **3549**, Springer Verlag, 2005, 148-172.

[9] **Gamma E., Helm R., Johnson R. and Vlissides J.,** *Design patterns elements of reusable object-oriented software,* Addison Wesley, 1994.

[10] **Samek M. and Montgomery P.Y.,** State oriented programming, *Embedded Systems Programming,* **13** (8) (2000), 22-43.
`http://www.embedded.com/2000/0008/0008feat1.htm`

[11] **Samek M.,***Practical Statecharts in C/C++,* CMP Books, USA, 2002.

[12] *OMG: MDA Guide*, OMG, June 2003.

[13] **Micskei Z. and Majzik I.,** Model-based automatic test generation for event-driven embedded systems using model checkers. *Proceedings of the International Conference on Dependability of Computer Systems (DEPCOS-RELCOMEX'06), Los Alamitos, CA, USA,* IEEE Computer Society, 2006, 191-198.

[14] **Pnueli A.,** The temporal logic of programs, *Proceedings of the 18th Annual Symposium on Foundations of Computer Science, 1977*, 46-57.

[15] **Sowmya A. and Ramesh S.,** Extending statecharts with temporal logic, *IEEE Transactions on Software Engineering*, **24** (3) (1998), 216-231.

[16] **Drusinsky D.,** Semantics and runtime monitoring of TLCharts, *Electronic Notes in Theoretical Computer Science*, **113** (2005), 3-21.

[17] **Pintér G. and Majzik I.,** Automatic generation of executable assertions for runtime checking temporal requirements, *Proceedings of the 9th IEEE International Symposium on High Assurance Systems Engineering (HASE 2005), Heidelberg, Germany*, eds. M. Dal Cin, A. Bondavalli and N. Suri, 2005, 111-120.

[18] **Mahmood A. and McCluskey E.J.,** Concurrent error detection using watchdog processors - A survey, *IEEE Transactions on Computers*, **37** (2) (1988), 160-174.

[19] **Michel T., Leveugle R. and Saucier G.,** A new approach to control flow checking without program modification, *Proceedings of the 1991 International Symposium on Fault-Tolerant Computing (FTCS), Canada, 1991*, 334-343.

[20] **Lu D.J.,** Watchdog processors and structural integrity checking, *IEEE Transactions on Computers*, **C-31** (7) (1982), 681-685.

[21] **Pataricza A., Dobán O. and Szőke Á.,** Costs and benefits of using formal methods, *Supplemental Volume of the 2004 International Conference on Dependable Systems and Networks, (DSN 2004), Florence, Italy, 2004*, 104-105.

[22] **Pintér G. and Majzik I.,** Model based automatic code generation for embedded systems, *Proceedings of the Regional Conference on Embedded and Ambient Systems (RCEAS-2007), 2007.*

[23] **Ansaldo Segnalamento Ferroviario S.p.A., CNR - Istituto di Scienza e Tecnologie dell Informazione, Budapest University of Technology and Economics, AZD Praha s.r.o. and Aalborg University Center for TeleInFrastruktur,** *Safe driver machine interface (DMI) for ERTMS automatic train control, Deliverable D2.2, Detailed HW and SW Specification,* EC Contract id.: IST-2006 - 031413, 2008.

[24] **Pataricza A., Majzik I., Huszerl G. and Várnai G.,** UML-based design and formal analysis of a safety-critical railway control software module, *Formal methods for railway operation and control Systems*, eds. G. Tarnai and E. Schnieder, L' Harmattan, Budapest, 2003, 125-132.

G. Pintér, Z. Micskei and I. Majzik
Department of Measurement and Information Systems
Budapest University of Technology and Economics
{pinterg, micseiz, majzik}@mit.bme.hu