VERIFICATION OF UML STATECHART MODELS OF EMBEDDED SYSTEMS

Ádám Darvas and István Majzik
Budapest University of Technology and Economics,
Dept. of Measurement and Information Systems
Magyar Tudósok krt. 2.
H-1117 Budapest, Hungary
da211@hszk.bme.hu, majzik@mit.bme.hu

Balázs Benyó
Széchenyi István University,
Department of Informatics
Egyetem tér 1.
H-9026 Győr, Hungary
benyo@szif.hu

Abstract. *This paper presents a method for verification of UML statechart models generated in the development process of embedded systems containing hardware and software components. The method allows the automated verification of behavioral requirements through model transformation and application of an off-the-shelf model checker. The transformation tools have been implemented and the method was applied successfully in the design verification of an interrupt controller. The paper deals with the details of the verification method and introduces the application example as well.*

1 Introduction

The test and verification are essential phases of any system development. With the enlargement of the size and complexity of the system under development the importance of these tasks increases.

Traditionally the system development is divided into *analysis & design* and *implementation* phases regardless of the development process applied (Fig. 1).

The system development process is typically accompanied by the occurrence of design and implementation faults. To avoid these faults the implementation phase is followed by a *test* phase. The test phase is intended to recover the differences between the *design* and the *implemented system*, i.e. to find the faults that arise in the implementation phase.

The activities of the test phase include *test case definition* and *test case execution*. The test phase can substantially be accelerated by the automation of these activities. The automation of test case execution on the basis of the design is often supported by CASE tools [1]. The automated definition of the test cases could be implemented only in the case when the design itself is described by a formal model [3,4]. These automated tools are essential in the development of large systems since they can efficiently accelerate the execution of the time-consuming testing.

The faults of implementation can be discovered by the test phase, however, the faults of the design may remain undiscovered. In order to recognize these faults, the development process has to be completed by the *design verification* phase. The goal of design verification is to find the faults in the design itself. The fault may be instantiated as an *inconsistent design* or as an *inequivalence of the design with the original requirements*. In the case when the requirements are not defined formally, the equivalence of the requirements and the design can be verified only manually.

In the modern system development process (supported by CASE tools) the design is generated in the form of a formal description. In this case the checking of the design can be supported by automated tools.

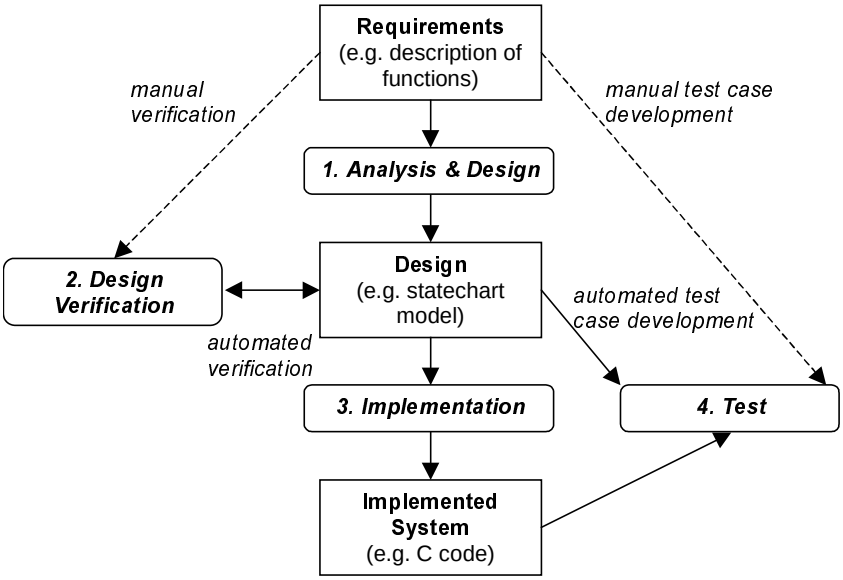

Figure 1: The relationship of the different phases of system development

The goal of this research was to develop a method for verification of designs described in the form of UML statechart diagrams.

Statechart diagrams were selected because statecharts are a widely used modeling language of embedded systems containing digital hardware and software components. Statechart diagrams are used traditionally in the field of embedded system design and in the process automation [5,2]. Also the designers of communication systems use statecharts extensively (e.g. in protocol design) for the description of the behavior of communicating objects.

2 Automated verification of embedded systems based on model transformation

Concurrency and distribution are common principles in embedded system design. In these systems multiple, potentially independent stimuli result in concurrent threads of execution. These threads can be deployed on multiple processors (microcontrollers, intelligent sensors and actuators). Concurrency and distribution are also used to increase performance and controllability.

Nowadays the object-oriented (OO) paradigm is becoming more and more popular in concurrent system design. Active objects represent (independent) threads of control, passive objects represent data structures used in the computation. In embedded systems, the objects (especially objects modeling hardware components or interacting with the environment) communicate through signals. In OO terms, a state transition in an object results in a send action, which raises a signal. Another object, through an event queue, receives the signal which triggers a state transition (and thus possible other actions).

Designing concurrent OO systems usually necessitates a thorough verification of concurrency control, i.e. synchronization and communication. Most problems with concurrent software arise from the coordination of interactions, which may result in deadlocks or other unwanted (hazardous) states. As the complexity of the systems increases, the manual verification of these properties becomes more and more error prone. Automatic support is required to help the designer in this task.

The semi-formal design languages used in OO system design open a new possibility of automated verification. Diagrams of UML, the standard description language of OO systems, can be transformed to mathematical models amenable to formal analysis [6].

We have elaborated and implemented a tool set which is able to transform UML models of concurrent OO systems to the input format of the model checker SPIN [7]. SPIN is a widely used tool for analyzing the logical consistency of distributed systems. We selected this tool since its formal modeling language called Promela (Process Meta Language) allows for the creation of concurrent processes (threads of control), and communication can be directly modeled by using both synchronous and asynchronous (i.e. buffered) channels. Using SPIN, the designer can check the concurrency control in his design relatively easily. We will present the possibilities in Section 4. Note that timing-related properties of the design (e.g. deadlines, time required for processing) can not be checked by SPIN.

2.1 Theoretical basis and extensions

The behavioral diagrams of UML, i.e. the statechart diagrams are transformed to Promela in two steps.

In the first step, each statechart diagram is transformed to a semantically equivalent formal model called Extended Hierarchical Automata (EHA). EHA can be considered as a formal syntax of UML statecharts, describing the statechart elements in a concise format, resolving the problem of inter-level and composite transitions by using special labels (Fig. 2).

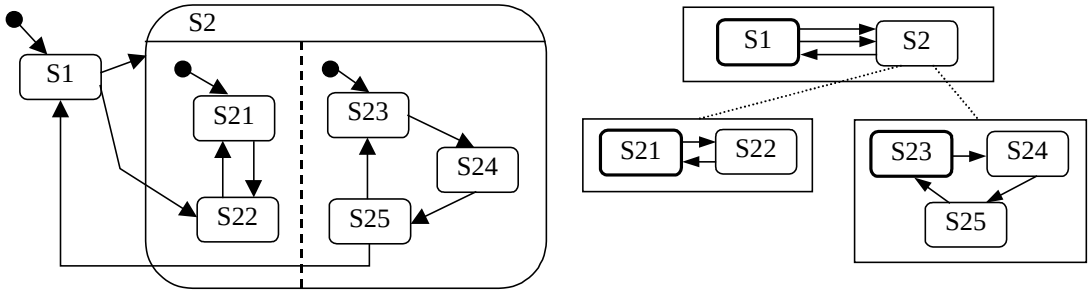

Figure 2: An UML statechart and the structure of the corresponding EHA (the concurrent regions in the refinement of state S2 are represented by two separate sequential automata in the EHA)

Since UML is a semi-formal notation, the definition of a formal semantics of UML statecharts is necessary to enable automatic formal verification. The formal operational semantics of the EHA representation is defined in the form of Kripke structures in [8]. The rules used to map UML statecharts to EHA are introduced in the same paper.

In the next step, based on the formal operational semantics, the EHA is transformed to Promela. The rules of this step were introduced and formally proved in [9].

These results allowed us to verify a single statechart (i.e. the behavior of a single object) only. We extended this approach to cover multiple statecharts, i.e. multiple objects of a distributed system, communicating through event queues. Moreover, in our system objects can be generated dynamically (however, the number of the objects has to be limited by the designer), and the deployment (i.e. event queue) of an object can be changed dynamically.

The UML semantics introduces the notion of event queue and event dispatcher; however, it does not describe how to specify them in the model. Accordingly, we had to define a set of modeling conventions:

- Event queues are modeled by instances of stereotyped UML classes with attributes defining the size of the queue and the policy of the dispatcher (FIFO or set).
- Objects belonging to the same event queue are grouped in packages.

These extensions enable a flexible configuration of objects according to the deployment and the needs of the operating system/run-time environment targeted by the design.
According to the conventions, in the formal model each EHA is assigned to an event queue belonging to the package of the corresponding object. Targets of send actions are specified by named objects, send actions without target result in broadcast events.

2.2 Implementation of the model transformation

The implementation follows the two steps described in the theoretical basis of the transformation (Fig. 3).

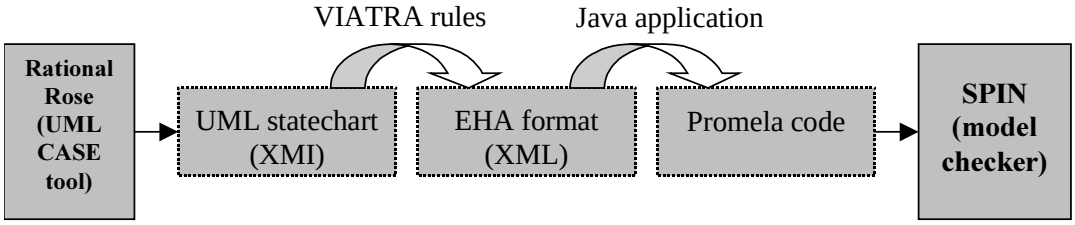

Figure 3: Implementation of the model transformation

Since both UML and EHA are (mainly) graphical languages, the transformation from UML statecharts to EHA was implemented by the graph transformation based tool set VIATRA [10]. The UML model elements are exported from the CASE tool in XML format (standard XMI) and graph transformation rules generate the model elements of the EHA:

- The structure of an EHA corresponding to an object is determined by the statechart of the parent class of that object.
- An EHA is composed of simple sequential automata related by a state refinement function, as determined by the state refinement relation in the UML statechart (Fig. 2).
- UML states are mapped to EHA states directly.
- Each UML transition is mapped to a unique transition in the automaton that contains all the source and target states. In case of compound and inter-level transitions, the original source and target states are included in the label of the transition.
- Labels of transitions contain the original source and target, the trigger event, the guard and the action of the transition.
- Targets of send actions are linked to event queues.

In the second step, the XML representation of the EHA is post-processed by a Java application that generates the Promela code [11]. Each EHA (i.e. object) is represented by a separate Promela process that expands the tree-like structure of the state refinement. The extensions of the basic approach described in [9] can be summarized as follows:

- Each event queue is modeled by a Promela data structure: A set is implemented by a set of bit variables, a FIFO queue is represented by a channel. The dispatcher is a separate Promela process that forwards the events to the target processes.
- The ordering of events in FIFO queues is sensitive to the ordering of the actions that generate them. The non-deterministic execution order of concurrent actions is modeled by concurrent sub-processes.

- Limited number of object creations (calls to constructors) is modeled by starting new Promela processes.
- System reconfiguration (i.e. changing the deployment of an object by a distinguished action of another object) is modeled by dynamically linking the moved object to another event dispatcher.

The extensions implemented in the transformation enable the verification of reconfigurable distributed object-oriented designs.

3 The interrupt controller example

A real example has been selected to demonstrate the application of the design verification method. The system under development is designed to control an ASI Master card [4]. The ASI is a field bus standard used in process automation. The software was designed to control the Master node of the field bus network.

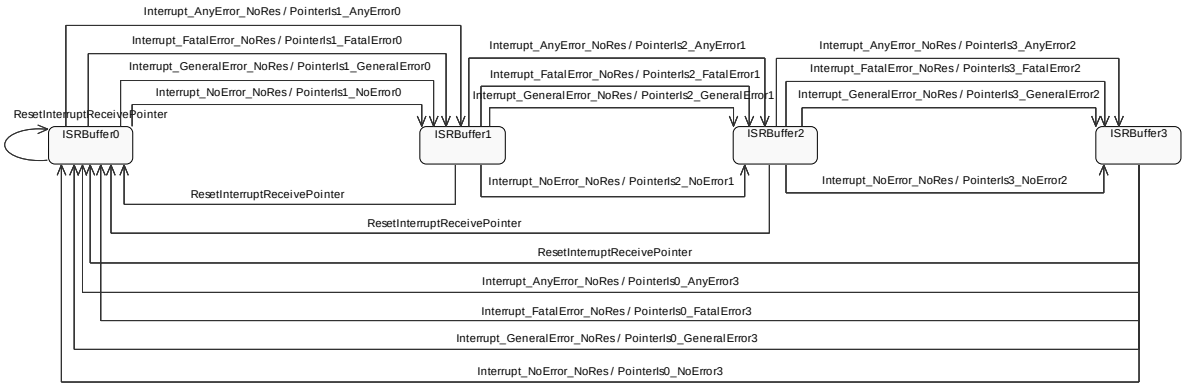

Figure 4: Statechart of the interrupt controller of the ASI Master

The ASI Master Firmware is an interrupt driven system. In the example, the behavior of its interrupt controller is verified (together with the behavior of its environment that generates the signals). The statechart diagram of the core behavior of the interrupt controller is presented in Fig. 4.

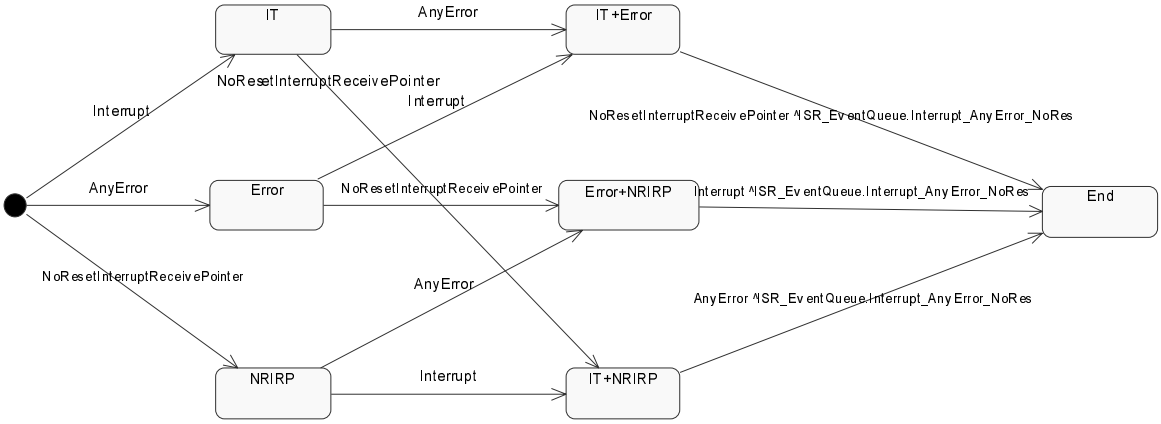

Figure 5: Statechart diagram of the event generation

The state transitions are initiated by one or more events and could result in the generation of another events. The dependencies are modeled by statechart diagrams as well. In Fig. 5 an example of this conditional event generation is presented. The condition of the generation of the event *Interrupt_AnyError_NoRes* is the occurrence of the following three events: *NoResetInterruptReceivePointer*, *AnyError*, and *Interrupt*.

The model of the full system consists of 7 statechart diagrams containing 39 states and 82 transitions.

4 Design verification

4.1 Checking correctness and reachability properties with SPIN

There are two ways of using SPIN for verification: the system model specified in Promela can be simulated or the fulfillment of formalized requirements can be examined.

A possible execution of the system can be examined during simulation by tracking the communication between the processes as well as the status of variables and channels.

During verification SPIN checks automatically for deadlocks and unexecutable code. Further requirements can be expressed by inserting special labels in the Promela code of the system.

Valid end states can be marked with the *end* label. That is necessary for two reasons. From one hand if the examined model reaches an end state (i.e. there is no further state transition that is enabled) the tool has to be able to decide whether it means deadlock or it is a proper end state of the system. Labeling a state with the *end* label tells the tool that it is the latter case. From the other hand, the examined model may wait in a cycle, so it does not reach an end state. Again, that may be a failure or the intended functionality of the system that should be distinguished. Labeling one of the states of the cycle indicates that it is not an error.

Progress and *accept* labels are used to establish the validity of cycles as well. An infinite cycle is valid if it passes through one or more states labeled with the *progress* label. The *accept* label means just the opposite: a state with the *accept* label must not be passed infinitely often.

Using these labels, the examined system can be analyzed for correctness violations. Moreover, temporal properties can be checked by the special *never* process which runs parallel (synchronized) with the other processes and may match any system state (but it is not allowed to modify them). The termination of the *never* process expresses undesirable or illegal behavior, moreover, the code of this process may also contain labels.

To compose the special behavioral requirements the designer does not need to have knowledge about the labels or the *never* process as SPIN provides automatic translation from Linear Temporal Logic (LTL) formulae into the *never* process. An LTL formula may contain usual Boolean operators and predicates as well as the following temporal operators:

- [] P: „P is *always* true during execution”,
- <>P: „P will be *eventually* true during execution”,
- P1 U P2: „P1 is true *until* P2 evaluates true”.

The verification is performed by exhaustive state space search, that is SPIN examines every possible sequence of execution. If any kind of behavioral violation occurs (i.e. deadlock, terminated *never* process etc.) SPIN stops the verification and starts to run a simulation showing the violating execution.

4.2 Verification results of the example

The transformation of the UML model of the example required less than 8 seconds on a common personal computer (300MHz Pentium-II processor, Windows NT 4.0 operating system, 128 Mbytes of RAM). The resulting Promela code contains 7 processes (one for each statechart), in addition to the event dispatcher and the initialization. The length of the Promela code is 416 lines.

The first verification step targeted the checking of unreachable code resulting from design errors. Among others, inconsistent use of the names of trigger events and send actions could be checked in this way. A typo in the event name that resulted in the unreachability of the state *ISRBuffer3* was discovered in less than 2 seconds by an exhaustive search covering 295 states and 306 transitions. The system entered an infinite cycle.

The results of the counter-examples are available as message sequence charts. In Fig. 6 a small fragment of the MSC illustrating the startup of the system is presented (the dispatcher forwards the first event to the processes).

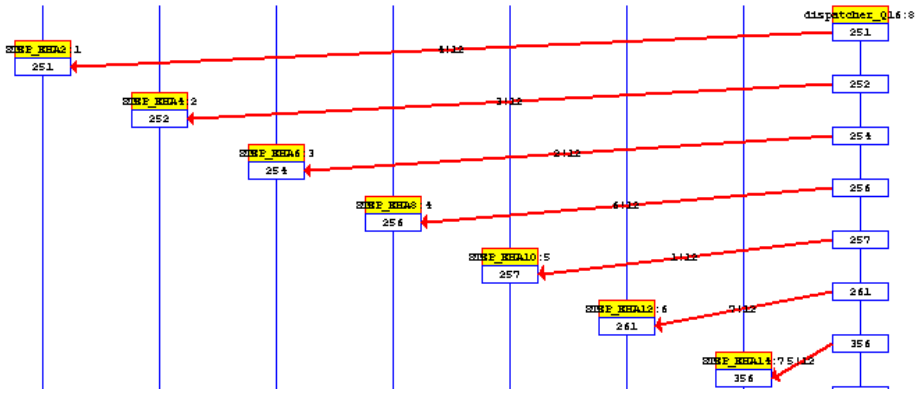

Figure 6: Startup of the system as presented by the MSC in SPIN

The verification of invalid end states proved that there is no deadlock in the system.

In the next phase of verification, the reachability of some selected state configurations (i.e. combination of active states in separate components) was initiated. The reachability of state combinations can be expressed in linear temporal logic using the <> (eventually) operator in SPIN and translating the expression to a *never* process automatically.

5 Conclusions

The paper presented an automated verification process of embedded systems designed by using UML statechart diagrams. The method extends the previous results and allows the analysis of distributed (and even reconfigurable) object-oriented systems. The verification is performed by an automatic model transformation from the UML models to a widely used verification tool. The transformation preserves the behavioral properties of the system.

The approach can be used to avoid logical design errors in a (relatively) early design phase, before the implementation, test generation, and execution begins. The applicability is constrained by the exhaustive nature of the verification: the state space explosion in highly parallel systems may prevent the designer to check complex system-level properties. In this case simulation or analysis reduced to core critical parts of the system can be considered.

Acknowledgments

The research was supported by the Hungarian National Research Fund (grant No. OTKA-F029739) and by the Hungarian Ministry of Education (grant No. FKFP 0200/2001 and FKFP 0103/2001). The research work of Balázs Benyó is supported by the Hungarian Ministry of Education (Békésy György Scholarship).

The UML to EHA transformation was implemented by Dániel Varró (Budapest University of Technology and Economics) in the VIATRA environment. His help is greatly appreciated.

References

1. P. Várady, B. Benyó: A Systematic Method for the Behavioural Test and Analysis of Embedded Systems, INES 2000, 4th IEEE International Conference on Intelligent Engineering Systems 2000, Sept 17-19, Portoroz, Slovenia, pp.177-180
2. P. Várady, B. Benyó, Z. Benyó: An Open Architecture Patient Monitoring System Using Standard Technologies, IEEE Transaction on Biomedical Communication Systems, 2001 (accepted paper - , reference number # 00_025)
3. B. Benyó: Verification of complex object oriented systems, 4th IEEE International Workshop on Design and Diagnostics of Electronic Circuits and Systems 2001, IEEE DDECS 2001, Győr, Hungary, April 18-20, 2001, pp. 289-290
4. P. Várady: Konzeption und Entwicklung einer Analysebibliothek zum Test des Verhaltens eingebetteter Software, Diploma Thesis in German, FZI-MRT Karlsruhe, 1997
5. D. Bursky: Embedded-Controller Architectures Suit All Needs. Electronic Design, January 8, 1996, pp. 53-64.
6. Bondavalli, M. Dal Cin, D. Latella, I. Majzik, A. Pataricza, G. Savoia: Dependability Analysis in the Early Phases of UML Based System Design. International Journal of Computer Systems - Science & Engineering, Vol. 16, No. 5, September 2001, pp 265-275
7. G. J. Holzmann: The Model Checker SPIN. IEEE Trans. on Software Eng, Vol. 23, No. 5, 1997, pp. 279-295
8. D. Latella, I. Majzik, M. Massink: Towards a Formal Operational Semantics of UML Statechart Diagrams. In P. Ciancarini, A. Fantechi, R. Gorrieri, (editors), Formal Methods for Open Object-Based Distributed Systems (Proc. FMOODS'99, Florence, Italy), Kluwer Academic Publishers, 1999, pages 331-347
9. D. Latella, I. Majzik, M. Massink: Automatic Verification of UML Statechart Diagrams using the SPIN Model-Checker. Formal Aspects of Computing, Vol. 11, No. 6, Springer Verlag, Berlin, 1999, pp. 637-664
10. D. Varró, G. Varró and A. Pataricza: Visual Graph Transformation in System Verification. In: E. Gramatova, H. Manhaeve and A. Pawlak (eds.): DDECS 2000 Design and Diagnostics of Electronic Circuits and Systems, Institute of Informatics, Slovak Academy of Sciences, Bratislava Slovakia, April 2000, pp. 137-141
11. Á. Darvas: Verification of distributed object-oriented systems. Technical report (in Hungarian). Student Paper Contest of the Budapest University of Technology and Economics, Faculty of Electrical Engineering and Informatics, 2001.