Completeness and Consistency Analysis of UML Statechart Specifications

Zs. Pap, I. Majzik[1], A. Pataricza and A. Szegi

Dept. of Measurement and Information Systems
Budapest University of Technology and Economics
H-1521 Budapest, Műegyetem rkp. 9.
Fax: +36-1-4634112
[papzs,majzik,pataric,szegi]@mit.bme.hu

Abstract. This paper describes methods and tools for automatic safety analysis of UML statechart specifications. Two types of analysis are presented. The first one checks completeness and consistency based on the static structure of the specification, thus it does not requires the generation of the reachability graph. Accordingly, this method scales up well to large systems. The second one performs dynamic analysis by checking safety related reachability properties with the help of a model checker. It is restricted to core critical parts of the system. Two approaches of the implementation of the static checking are discussed. The use of the tools is presented by a case study.

1 Introduction

Our life relies more and more on the dependability (especially safety) of embedded computer control systems. As the complexity of these systems increases, the task of the engineers in specifying, designing and validating the system becomes increasingly difficult. The need for efficient design has triggered the development of well-specified and standardized design methods and languages. Such languages should satisfy multiple requirements. First, designers ask for a general-purpose language which is easy to understand, close to their way of thinking, and has clearly arranged (self-documenting) visual models covering various aspects of the system. Second, validation and formal verification of the design require precise syntax and well-defined semantics in mathematical terms. Third, the development of integrated, computer-assisted design environments demands languages which are easy to process, navigate and manage. The recently created Unified Modeling Language (UML) [1] is a language designed to satisfy most of these requirements. It is a general-purpose object-oriented modeling language used to specify, visualize, construct and document different aspects of systems. UML is expected to become a de-facto standard for the design of systems from small embedded controllers to large and complex distributed systems.

UML can be used to construct software specification of embedded (real-time) systems [2], often implementing safety-critical functions. The specification is usually

[1] Supported by the Hungarian Scientific Research Fund under contract OTKA-F030553.

elaborated by the cooperation of users, domain experts and system engineers. Unfortunately, it is often incomplete, inconsistent and ambiguous. The errors in the specification are not only difficult and expensive to correct in the further phases of the life cycle, but also often lead to safety related failures. Accordingly, the early checking of the UML specification is crucial.

Our work aims at the elaboration of methods and tools for the checking of some aspects of completeness and consistency in UML models. We concentrate especially on the behavioral part of UML, namely the statechart diagrams. Statecharts are the most complex formalism used in UML (therefore errors occur most likely here) and have some specific features, like hierarchy and concurrency, which require non-trivial checking methods.

Our examination is focused on embedded control systems. In these systems, the controller continuously interacts with operators and with the plant by receiving sensor signals as *events* and activating actuators by *actions*. UML statechart formalism allows to construct a state-based model of the controller, describing both its internal behavior and the reaction to external events. Here we assume that the behavior of the controller is specified by a single statechart (unfortunately, standard UML does not specify the semantics of interacting objects precisely).

The paper is structured as follows. Section 2 is a short overview of the safety criteria and checking methods proposed in the literature. Section 3 outlines the main concepts of UML statecharts. Section 4 and 5 describe our work in static checking and reachability analysis, respectively. Our automatic tool is introduced in Section 6. The paper is closed by a short Conclusion.

2 Checking Completeness and Consistency Criteria

A safe system is free from accidents or unacceptable losses. Safety analysis should identify hazards based on the (formal) model of the system. Accidents related to computers are usually resulted from flaws in the specification (model). In [1] and [4] 47 formal criteria were defined that should be satisfied to avoid incorrect specifications. These criteria cover general aspects of the specification of a control system, including also peculiar ones like environmental capacity and data age.

The most important desirable properties of a specification are completeness and consistency. Completeness with respect to an embedded control system means that a response is specified for every possible input sequence, including also timing variations (early, lately, delayed etc. signals). Consistency of the specification implies that there are no conflicting requirements and no (unintentional) non-determinism.

Tool support for checking completeness and consistency is required, since manual checking is error-prone and time-consuming. From the point of view of the automated methods and techniques, different approaches can be distinguished:

- *Pure reachability analysis* of the state space of the system can detect ambiguous situations resulting e.g. from unreachable states, undesired global states (in concurrent models) or unwanted sequence of actions. The examination of the global state space requires the generation of the reachability graph, which often results in state space explosion.

- *Model checking* examines properties expressed in temporal logics. Modern model checkers try to handle the state space explosion by applying sophisticated methods in representing and analyzing the global state space (e.g. symbolic techniques, partial ordering).
- *Theorem proving systems* require to describe both the specification and the criteria in a formal logic, and prove that the criteria and the specification are suitably related.
- *Static analysis* is performed directly on the model and checks those criteria that are not related with the global state space.

The automatic tools proposed in the literature are in strict relationship with the formalism (language) intended to be checked, i.e. the complexity of the checking is heavily influenced by the formalism. In [5] a black-box system specification language RSML (Requirements State Machine Language) was proposed which enabled the automated checking of some criteria [6]. The latest development of the same authors is the experimental toolset SpecTRM [7] and the corresponding formal specification language SpecTRM-RL [8]. This language is designed especially to enforce the satisfaction of the safety criteria and enhance the ability to build tools that check them.

UML was developed without considering strict rules to force the designer to prepare a complete and consistent specification. The flexibility and extensibility of the language, and some of the applied constructs (like internal broadcast of events, which was identified as one of the typical sources of specification errors [8]) and semantic constructs (non-determinism) could make the hazard analysis of the specification of safety-critical systems difficult. However, it has to be pointed out that UML incorporates a technique to include static constraints on the usage of its model elements. OCL, the Object Constraint Language [9] is designed to specify well-formedness rules of the UML model (OCL expressions are used also in the language definition itself). By defining appropriate rules, well-formedness in the sense of completeness and consistency can be prescribed and then checked.

UML statecharts, as state-based specifications, can be subject of both static analysis and model checking (reachability analysis). Static analysis aims at the checking of general, application-independent criteria, while model checking is suitable to examine also application-specific requirements. The former checks mainly the proper static structure of the model, while the latter requires the formalization of the dynamic semantics of the language.

3 UML Statecharts

UML statecharts is an (object-oriented) variant of classical Harel statecharts [10]. The statecharts formalism itself is an extension of traditional state transition diagrams including the following additional concepts (explained with the help of the statechart presented in Figure 1):

- State hierarchy and concurrency. A state is called a *composite state* if it contains one or more substates (e.g. `Work` contains `Phase1`, `Task1`, etc.). A composite state can be decomposed into mutually exclusive disjoint substates or into orthogonal substates. In the latter case, the composite state is *concurrent* and its

direct substates are called *regions* (e.g. `Group1` and `Group2`). Regions must be further refined into substates (`Group1` is refined by `Phase1` and `Phase2`). States without further refinement, at the lowest level of the hierarchy, are called *basic states* (e.g. `Passed`).

- Compound transitions. A simple transition indicates that the system may change its state and perform a sequence of actions when a specified event occurs and a specified guard condition is satisfied (e.g. the transition from `Work` to `Failure`). Compound transitions have multiple segments. *Join segments* (e.g. to the state `Passed`) originate from concurrent regions (representing synchronization), while *fork segments* (e.g. from the state `Prepare`) are connected to concurrent regions (representing splitting of control). *Branch segments* labeled with guards compose different possible paths depending on conditions (e.g. the transition triggered by the `error` event).
- History states. A transition drawn to a history state is equivalent to a transition drawn to the *last active direct substate* of the composite state in which the history state resides (the circle with `H` in Figure 1). Firing of a transition drawn to a deep history state causes the last active substates of the composite state be entered recursively.
- Enriched set of events and actions. Events may have parameters. Actions are distinguished as call, return, send, terminate, create and destroy actions, according to the (object-oriented) software context.

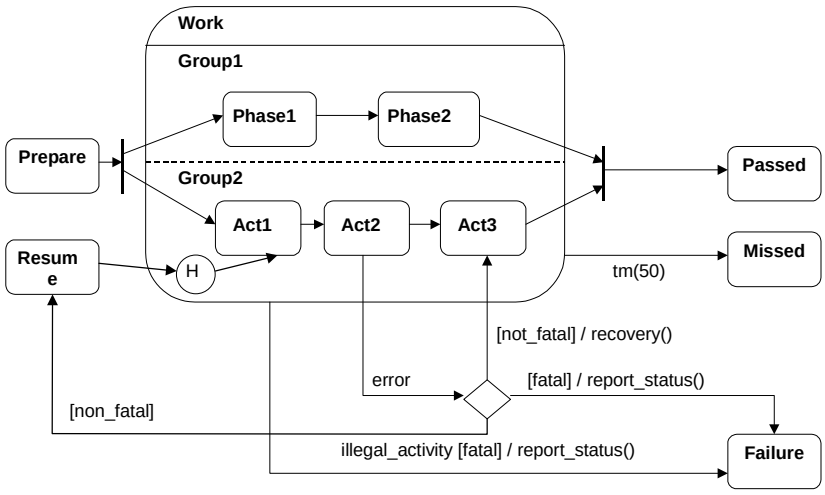

Fig. 1. An example statechart

The peculiarities of UML semantics can be summarized as follows:

- Single event processing. The hypothetical state machine which implements the statechart diagram processes event instances that are selected by a dispatcher from an event queue. Events are processed one by one, each after the other. Accordingly, transitions are triggered by at most one event.

- Run-to-completion processing. An event stimulates a run-to-completion step. Transitions that fire have to be fully executed and the state machine has to reach a stable state configuration before it can respond to the next event.
- Priority concept. Transitions are in conflict when the intersection of the sets of states they exit is non-empty. Some conflicts can be resolved by using priorities. A transition has higher priority than an other one if its source state is a substate of the source of the other one. If the conflicting transitions are not related hierarchically then there is no priority defined between them, and the conflict is resolved by selecting one of the transitions non-deterministically.
- Execution step. The set of transitions that will fire is a maximal set in which all transitions are enabled (i.e. they are triggered by the actual event, their guards are satisfied and their source states are active), there is no conflict within the set, and there is no enabled transition outside the set that has higher priority than a transition in the set. The firing order of transitions in the set is not defined.

4 Static Analysis

Completeness and consistency can be given as criteria to be satisfied by the hypothetical state machine (automaton) implementing the statechart specification. The global states of this automaton are called *statuses*, its transitions are called *step transitions*. Statuses are composed of active states of the statechart by considering that if a composite state (concurrent composite state) is active then one of its substates has to be active (in each of its regions, respectively). Step transitions are composed of (concurrent) transitions of the statechart. Completeness means that in all possible statuses of the automaton, for all possible events, there must be a step transition (in a special case an internal transition) defined which is triggered by the event. Consistency means that in each status, each event should trigger only a single step transition.

Static checking requires to re-formulate these criteria in syntactic terms of the statechart, taking into account the UML semantics covering hierarchy, concurrency, and priority scheme. In the following we examine states, transitions, guards and compound transitions of the statechart and formulate the criteria in these terms.

4.1 States and Transitions

The hierarchical structure of statecharts could be unfolded to a flat state diagram, but this representation would not make the checking easier (due to the increased size of the flat state diagram). Accordingly, we decided to check the hierarchical model directly. In this model, the basic states fully determine the status of the automaton.

Transitions of composite states are taken into account by considering that substates virtually inherit the transitions of their parent states. Implicit transitions and transitions from/to composite states are resolved as follows. A transition drawn to a boundary of a composite state means a transition to its initial substate or to the initial substates of its regions. A transition drawn from a boundary of a composite state means that all of its substates are exited when the transition is taken (fires). If a

transition enters a region of a concurrent state, then the other regions are also entered (explicitly by fork segments or by default entering their initial states). Similarly, if a transition exits a region of a concurrent state then all of the other regions are also exited.

When completeness is being checked, the basic states are examined and all inherited transitions are considered. Consistency requires the checking of individual states, since conflicts among transitions on different hierarchy levels are resolved by the priority scheme of UML. In this subsection, we consider transitions without guards (guards are discussed in the next subsection).

Completeness requires that in each basic state, for all possible events, there must be a transition defined. If an event should not have any effect in a given state, then the designer should define it as an event triggering an internal transition (that does not change the state) with no action. Note that self-loop transitions in UML are not suitable for this purpose, since firing of this kind of transitions induces the execution of the entry and exit actions of the state.

In a concurrent composite state, it is possible that a transition is defined only in one of the regions (remember that the status of the automaton is a composition of the basic states of the regions). The exploration of the possible combinations of basic states in the regions would need the generation of the reachability graph. To avoid this problem and allow the static checking, it is required that basic states in concurrent regions are defined by the designer in the same way as non-orthogonal states (i.e. the above criterion should be satisfied).

Completeness requires that each state is targeted by (at least one) transition. Note that initial states have an incoming transition from the initial pseudo-state.

Consistency of the specification requires that in each state, only a single transition is triggered by a given event. It means that there are no conflicts that are not resolved by the priority scheme of UML. Namely, conflicts among different hierarchy levels are always resolved as the transition with source state at the lowest level fires. Conflicts not resolved by the priority scheme, thus requiring non-deterministic choice, occur only at the same hierarchy level.

Another source of possible inconsistency is that the firing order of transitions enabled in concurrent regions is undefined. From the point of view of the environment, it may result in non-deterministic execution order of actions belonging to these transitions. The consistency criteria says in this case that there are no pairs of transitions in concurrent regions that may fire at the same time, and both have actions defined. Reachability analysis is required to check this criteria, since these aspects of concurrency (i.e. the possible statuses of the automaton) can not be checked by the static analysis.

4.2 Guards

Completeness requires that in each basic state, considering also inherited transitions, guards of transitions triggered by the same event form a tautology.

Consistency is checked by the following criterion: If there are two or more transitions that are originating from the same state and triggered by the same event, then their guards could not be true at the same time.

The specification is ambiguous if the designer utilizes the priority scheme of UML in a strange way: guards of transitions originating from a given state and triggered by the same event form a tautology, and at the same time there is a transition which is triggered by the same event and originates from a parent state. This latter transition will never fire.

Checking of these rules is difficult, since in UML guard conditions can refer to variables, functions, parameters of events, and orthogonal states (by *in_state()* predicates prescribing that some orthogonal states are active). Accordingly, the checking would require in worst case both reachability analysis and the interpretation of the program code assigned to actions. Static checking of guard conditions is possible only if the guard expressions are restricted and expressed in a special canonical form (as proposed in RSML and SpecTRM). Simple guards referring to constants can be checked more easily, this type of checking is built into our checker.

Completion transitions have no trigger events and are executed as soon as the source state is reached and the guard is true. They can be checked according to the above rules, i.e. (i) completeness requires that guard expressions of completion transitions originating from a basic state (considering also inherited transitions) form a tautology, and (ii) consistency requires that only one of them is true at the same time. *If there is a state in which the guards of a normal and a completion transition can be true at the same time then the specification is ambiguous, since in this case the normal transition will never fire.*

4.3 Compound Transitions

Compound transitions are transitions consisting of multiple segments like conditional branches, fork and join transitions. For the sake of the analysis the compound transitions are transformed to simple ones, that can be checked using the above rules.

Conditional branches are converted into separate simple transitions originating from the common source state and reaching the target state of each possible path of segments. The guard conditions are formed by the AND relation of the individual guards along the segments on the path from the source state to the target one.

Fork transitions consist of multiple segments originating from a state and reaching concurrent regions. They are converted into simple transitions in the same way as conditional branches, but the redundant transitions are marked at the source state.

Join transitions originate from concurrent regions and reach a single state. They are converted into a set of simple transitions that originate from each source state, reach the original target state, and inherit the original guard. A join transition can be taken only if all source states are active. This condition is expressed on the set of the newly generated transitions by additional *in_state()* guards referring to the other source states and being in AND relation with the above mentioned guard expression. Accordingly, the consistency checking of join transitions requires reachability analysis.

4.4 Classification of States

In embedded control systems the controller manages an internal model of the controlled system. Similarly, the operator has a mental model of the controller interface. The differences between these models, i.e. the semantic distance, should be minimized.

Immediately after controller startup or in exceptional situations the internal model is not synchronized with the state of the controlled system. Until the synchronization is performed (e.g. by reading sensor signals), the actions of the controller should be restricted. The designer should convince the checker that he/she considered this requirement by marking some (sequence of) states as *Unknown*. UML stereotypes, that allow a high-level classification of model elements, can be used for this purpose. Initial states should always be stereotyped as *Unknown*.

Accordingly, *completeness requires the existence of the* Unknown *stereotype in the initial state(s) of the controller's statechart*.

4.5 Time-out

If the controller stays in a specific state too long, then it could induce that the consistency between the controller and the controlled system has been lost (due to missing or unexpected events). The designer should handle this kind of exception by specifying a time-out transition, i.e. a transition triggered by a time event, from each status of the automaton. In UML, a time event can specify a trigger, which denotes the time elapsed since the state was entered. The time-out transition should lead to *Unknown* states, where the lost synchronization is restored.

Accordingly, *completeness requires that in each basic state of the statechart of the classes used as Models (considering the inherited transitions) there is a time-out transition leading to a state stereotyped as* Unknown.

Consistency requires that at most one time-out transition is found in each basic state (it is ambiguous if both a substate and its parent state have time-out transitions).

5 Reachability Analysis

Reachability analysis can check both general properties requiring the generation of the reachability tree (e.g. completeness criteria referring to join transitions mentioned in the previous section) and application-specific safety requirements (e.g. avoidance of unsafe statuses in concurrent specifications). We consider here model checking as a technique which covers the traditional reachability analysis.

A mandatory prerequisite for model checking is to map statechart diagrams to a formal semantics model. In a previous work a subset of UML statecharts was mapped to Kripke structures [11], and it was proved that the mapping satisfies the properties of UML semantics given informally in [12]. The subset does not consider dynamic object-oriented features like inheritance, creation and destruction of objects, but includes all aspects related to concurrency like sequentialization, non-determinism and parallelism. Accordingly, it is suitable to be used in our environment.

Based on the Kripke structure a translation to Promela, input language of the model checker SPIN [13] was also defined. SPIN was selected since it is one of the most efficient tools available, and Promela allows the specification of state variables, communication actions, and a variety of requirements. There are built-in capabilities to check deadlocks, invalid endstates, non-progress and acceptance cycles. Application-specific requirements can be given in the form of assertions (invariants inserted into the Promela code), a never claim (an automaton that defines a behavior that should not be matched) or linear temporal logic formula (among others invariance, response, and precedence properties).

SPIN helps in checking completeness as follows:

- Unreachable states (unreachable code) are reported automatically by SPIN.
- Missing transitions are detected by analyzing invalid endstates.
- Enabledness of join transitions can be checked by assertions.
- Tautology of guards can be checked by using assertions formed by the OR relation of the guards. In Promela, *in_state()* guards and Boolean logic formula referring to integers can be easily evaluated.

Theoretically, it is also possible to check consistency by inserting assertions that evaluate to false if two or more transitions are enabled at the same time. (SPIN has no built-in capability to report non-determinism.)

6 The Checker Tools

We have considered three approaches to implement the static checking of the completeness and consistency criteria in UML statecharts:

- Theoretically, the majority of the criteria can be expressed in the form of *OCL expressions* interpreted on the UML metamodel of statechart diagrams. The metamodel defines (in a form of class diagrams) the UML model elements like State, Transition, Event etc. and gives their possible relationships and the syntactic constraints. By enriching these constraints (called here well-formedness rules), a "safety-critical UML" sub-language can be defined. Since completeness and consistency criteria can be given as additional well-formedness rules, the approach fits very well to the semantics of UML. The implementation of the checking requires either deep integration with the CASE tool used by the designer or an external interpreter, which can examine the design with respect to the restricted metamodel. This approach was considered as a subject of our future work.
- It is a natural idea to formulate the criteria in a general *logic language*. We selected Prolog for this purpose. The Prolog expressions are interpreted on the standard database of the UML model elements given in the design. Accordingly, a well-scalable, flexible and general solution is provided.
- The (hard-coded) *direct implementation* of the checking is the less flexible, but has the best performance characteristics. Fortunately, most of the UML based CASE tools support a standard interchange format (eXtensible Markup Language Metadata Interchange, XMI). The checking can be based on this output.

In the following, we will report our experiences with these variants of the tools and compare their advantages and disadvantages. It can be mentioned that these variants

can also be considered as N-version programming [14] of the same problem, thus allowing a fault-tolerant implementation of the completeness and consistency checks. Accordingly, the trustworthiness of the analysis can be increased, which is an important factor e.g. from the point of view of an audit.

6.1 The Prolog Approach

The Prolog-based tool variant was implemented utilizing the environment developed in the framework of the HIDE project (High-Level Integrated Design Environment for Dependability, ESPRIT Open LTR No. 27439) [15]. In this environment, a commercial UML-based CASE tool (MID Innovator [16]) was included for user-end modeling.

To interface this tool with other analysis and evaluation tools, a database representation of the UML model was elaborated. The structure of this database corresponds to the structure of the UML metamodel. From our point of view this representation is especially useful, since it assures an easy navigation and searching in the UML model, which task is the crucial point of completeness checking. A commercial database manager was used to handle the model database. The static completeness and consistency criteria were formulated as Prolog program predicates. Logic programming is suitable for the compact definition and easy analysis of these criteria.

Prolog was extended with an interface toward SQL. Accordingly, Prolog questions are converted internally into SQL commands, which are executed by the database manager. Results of the database search are back-annotated by the interface again as Prolog predicates. Final result of checking is the detected set of errors and the generated warnings of the Prolog program.

The concepts are demonstrated by the following example:

```
compare(STATEID,EVENTNAME,STATENAME):-
 (\+ trans(STATEID, TRANSITIONID, EVENTNAME, LEVEL, ISLEAF,
    GUARDBODY, GUARDEVAULATED, LEAFSTATEID)),!,
 format("~n-Error: Transition not specified in state:",[]),
 write(STATENAME), format(" Event:",[]), write(EVENTNAME).
```

This code segment is one of the definitions of the expression `compare`, which will find unspecified transitions. In the first row transitions fitting to `STATEID` and `EVENTNAME` are found. Here `trans` is an SQL view of the Transition and ModelElement tables of the database. It is processed by the SQL interface, which provides a dynamic knowledge base for Prolog. The second row is activated only if the first fitting was empty. If there is no transition with the specified state and event then an error message is produced.

The expression `compare` is used as follows:

```
check(STATE, NAME):-
        eventnames(EVENTNAME),
        compare(STATE, EVENTNAME, NAME),
        fail.
check(STATE, NAME).
```

Here `eventnames` is a set of dynamic predicates containing the names of all events. This function enumerates the events and compares it with the parameter `STATE`. It is used as follows:

```
check_model:-
        leafs(STATE, NAME),
        format("~nChecking State:",[]), write(NAME),
        check(STATE, NAME),
        fail.
check_model.
```

where `leafs` is again an SQL view referring to the set of basic states.

The time requirements of the checking are high, since (i) Prolog is an interpreted language, (ii) communication with the database manager is time-consuming by using relatively slow network connections, (iii) the Prolog-SQL interface is not optimized, and (iv) the model database is very fragmented (over 120 tables). The tool was optimized by implementing table joins in SQL. After this optimization the speed of the program has increased up to 200%.

Since the database server can handle extremely large tables, it does not limit the size of the model to be checked in a single step. On the other hand, the Prolog program must access and transfer this amount of information via local area network. Thus, if the model is large, this process can be very slow. Moreover, the Prolog program uses dynamic predicates from several tables, which requires a great amount of memory (in addition to the Prolog interpreter which is itself a memory-consuming program).

Main advantages of the Prolog approach are as follows:

- it is easy to formalize and implement the criteria (including also possible domain-specific additional rules),
- it is easy to read, understand and verify the Prolog program,
- the checker tool is portable (machine and operation system independent).

Unfortunately, in the case of large models the Prolog implementation is extremely slow. For instance the verification of the example model (see Section 7) needs more than 10 seconds. Of course, this time depends also on the speed of the local area network.

6.2 The Direct Approach

The model representation used by the second variant of the checker tool is the XMI compliant output of the UML CASE tool. XMI (eXtensible Markup Language Metadata Interchange) was standardized by the OMG in order to provide an easy interchange of metadata among tools using UML as their modeling language [17]. It integrates the metamodel architecture, UML and XML (Extensible Markup Language). Accordingly, the XMI compliant output consists of the standard Document Type Definition (DTD) file corresponding to the UML metamodel and the XMI document in XML format.

When processing the XMI output, our parser is based on the UML DTD definition. After processing, the UML model itself is loaded into the memory and the checker procedures (written in Visual C++) are executed.

The loader processes the behavioral part of the UML metamodel, builds the corresponding data structure in the memory and fills it with data from the XMI document. All elements of the XMI file are represented by objects. These objects are linked together into a hierarchical tree structure by using linked lists. The basic objects like ModelElement, Action, Signal, etc. and the classes of the BehavioralElements are implemented by special child classes, the others are loaded as general XMI elements. The special classes implement the metamodel of the Statechart, the derived objects can load the attributes of the metamodel. The other XMI elements read only the standard XMI attributes like ID, UUID, idref, etc. The cross-links are realized by textual references (idrefs). It would be possible to use pointers for this purpose, but in this case the loader could not process the XMI file in one step.

Essentially, the XMI loader builds the internal representation on which the checker procedures are executed. Typically, these procedures are recursive ones. If a statechart is embedded in another one (e.g. in a composite state), the checker procedure can automatically examine this as a part of the "parent" statechart. Naturally, two independent statecharts must be verified independently.

The checker procedures run noticeably faster than the Prolog-based implementation. The verification on the same example model (described in the next section) needed only 2 seconds, including also the time required to process the XMI file. Note that in the Prolog approach, the time required to generate the common database, which is again a few tens of seconds, was not included. If we converted all textual references into direct pointers, the verification process could be even faster.

In a typical UML model there are less than 1000 states [6]. An XMI object representation needs about 60-100 bytes in the memory. Accordingly, the full model could be about 10 MB large. This means that a standard model can fit into the memory of a common PC.

This variant of the tool is fast and efficient, but it has a few disadvantages as follows:

- The model representation is less scalable and robust than that of a general purpose database manager.
- The implementation of the checker procedures is more complex than in the case of the Prolog version.
- The portability of the tool is problematic.

Theoretically, the database approach could support teamwork (when designers work on different parts of the same model concurrently) very well. This is not possible in the current implementation of the direct approach.

We can also combine some advantages of the direct and the Prolog/SQL-based approaches. The tool-dependent database export is replaced by a loader that fills the database structures by processing the XMI model output. This method behaves similarly to the Prolog approach.

7 Case Study

Our work on the completeness and consistency checking was motivated partially by our experiences gathered during the design of a safety-critical, embedded real-time

system, a fire-alarm backup controller (referred to as VE in the following) which is part of a complex fire/gas/security alarm system.

The VE is a complex unit, its software model has more than 50 classes and 60 modules, and the program implementation itself is longer than 30,000 rows in C. The operating platform is an embedded microcontroller, which made the testing and debugging difficult.

The original version of the VE software was created by conventional programming techniques based on a natural language specification. After the implementation the testers tried to examine the most important scenarios, but of course not all possible cases could be tested (as asynchronous input signals are processed by the unit).

The unit was put into operation and, unfortunately, hard-to-check intermittent failures were detected. On average once in every week (after a few millions of correct polling cycles) the module of the system responsible for the communication came to an erroneous state for 3-4 seconds. During this time the communication was broken, the control station generated an error alarm and for a few minutes the fire protection was disabled, which resulted in a hazardous situation. Since this software problem occurred randomly, the thorough testing and debugging of the problem seemed to be hopeless.

The decision was to re-implement the module from the beginning. The natural language specification was formalized in UML and the model was checked also for the sake of completeness and consistency.

On Figure 1 part of the UML statechart specification of the serial communication controller module of the VE is presented. (The shaded rounded rectangles represent stable states, while the others are temporary states.). This module controls two independent serial ports: one for the data-collector units (CVKE) and one for the central station. When the central station polls the CVKEs, the VE forwards the polling commands and the responses (this is called “Transparent Mode”). If the central station does not send polling commands for a predefined period of time (here 3 seconds), the VE starts an autonomous operation (called here “Master Mode”) and starts sending the polling commands itself. If the central station resumes the polling then the VE must switch back to Transparent Mode.

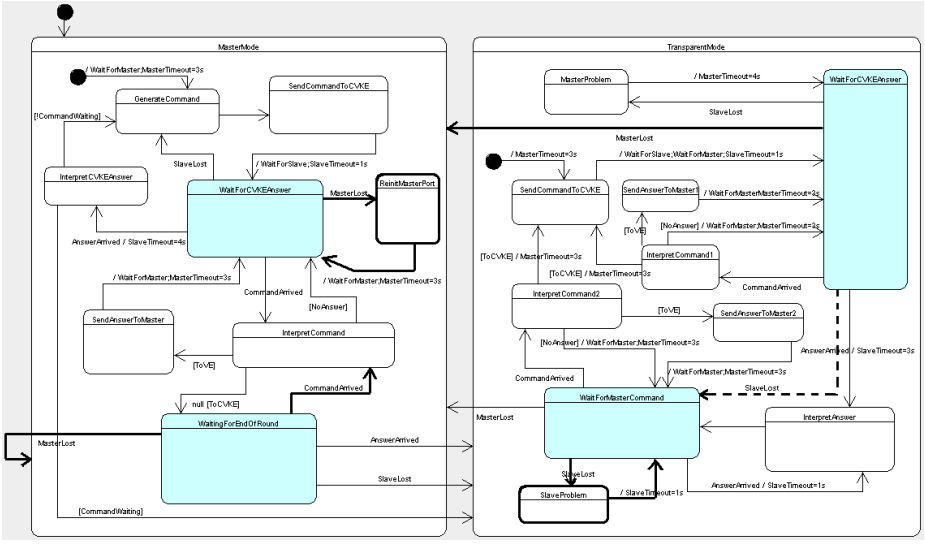

Fig. 2. Statechart of the serial communication module

The statechart model of the VE was checked by our completeness and consistency checkers. Several deficiencies were detected. For example, considering the statechart depicted in Figure 1, the following problems were identified (shown on the figure by thick lines):

- There was no transition specified for the Timeout condition “MasterLost” in state “MasterMode:WaitForCVKEAnswer”.
- There was no transition specified for the Timeout condition “MasterLost” in state “MasterMode:WaitForEndOfRound”.
- There was no transition specified for the Timeout condition “MasterLost” in state “TransparentMode:WaitForCVKEAnswer”.
- There was no transition specified for the Timeout condition “SlaveLost” in state “TransparentMode:WaitForMasterCommand”.
- There was no transition specified for the event “CommandArrived” in state “WaitForEndOfRound”.
- In State “TransparentMode:WaitForCVKEAnswer”, event “SlaveLost” triggered two transitions (one of them is shown by dashed line) resulting in a potentially non-deterministic operation.

We have performed also some additional completeness and consistency checks, which are specific for this control system. For instance we had to check whether for all scenario, the Timeout Timer is re-started in every stable state. The reachability analysis was performed completely in the HIDE environment since the mapping of UML statecharts to Promela code was implemented there.

The full verification and the correction of the specification of the communication module required approx. 4 hours. Then the skeleton of the program code was generated automatically, based on the checked UML model. The finalization of the code was made manually.

The new code was tested, integrated and the system was put into operation. The intermittent failures disappeared from the system. The problems were presumably caused by the incompleteness of the specification in the case of Timeout events.

7 Conclusion

The paper presented methods and tools for the checking of some aspects of completeness and consistency in UML statechart specifications of embedded controllers. Criteria were formulated and their checking was proposed both by applying static methods and reachability analysis (model checking). The main contribution of our work are (i) the adaptation of existing criteria to the UML statechart formalism and (ii) the experimental implementation of the checker methods. This work can be considered as a first step towards the automatic analysis of the majority of criteria given in [4].

Our further research concentrates on two main topics. The first goal is the extension of the set of criteria to be checked. The next area of research is the investigation of the use of OCL to express and check safety criteria. Finally the research could lead to the definition of a *sub-language of UML* suitable for specifying safety-critical systems.

References

1 J. Rumbaugh, I. Jacobson and G. Booch: The Unified Modeling Language Reference Manual. Addison-Wesley (1999)

2 B. P. Douglass: Real-Time UML - Developing Efficient Objects for Embedded Systems. Addison-Wesley (1998)

3 M. S. Jaffe, N. G. Leveson, M. P. E. Heimdahl, and B. E. Melhart: Software Requirements Analysis for Real-Time Process-Control Systems. IEEE Trans. on Software Engineering, Vol. 17, No. 3, pp 241-258 (1991)

4 N. G. Leveson: Safeware: System Safety and Computers. Addison-Wesley (1995)

5 N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese: Requirements Specification for Process-Control Systems. IEEE Trans. on Software Engineering, pp. 684-706 (1994)

6 M. P. E. Heimdahl and N. G. Leveson: Completeness and Consistency Checking of Software Requirements. IEEE Trans, on Software Engineering, Vol. 22. No. 6 (1996)

7 N. G. Leveson, J. D. Reese and M. Heimdahl: SpecTRM: A CAD System for Digital Automation. Digital Avionics System Conference, Seattle (1998)

8 N. G. Leveson, M. P. E. Heimdahl, and J. D. Reese: Designing Specification Languages for Process Control Systems. Lessons Learned and Steps to the Future. http://www.safeware-eng.com/pubs/

9 Rational Software, Microsoft, Hewlett-Packard, Oracle, Sterling Software, MCI Systemhouse, Unisys, ICON Computing, IntelliCorp, i-Logix, IBM, ObjecTime, Platinum Technology, Ptech, Taskon, Reich Technologies, and Softeam: Object Constraint Language Specification, version 1.1, (1997)

10 D. Harel: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming, Vol. 3, No. 3, pp 231-274, (1987)

11 D. Latella, I. Majzik, and M. Massink: Towards a Formal Operational Semantics of UML Statechart Diagrams. In P. Ciancarini and R. Gorrieri, editors, IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Oriented Distributed Systems, Kluwer Academic Publishers (1999)
12 Rational Software, Microsoft, Hewlett-Packard, Oracle, Sterling Software, MCI Systemhouse, Unisys, ICON Computing, IntelliCorp, i-Logix, IBM, ObjecTime, Platinum Technology, Ptech, Taskon, Reich Technologies, and Softeam: UML Semantics, version 1.1, (1997)
13 G. Holzmann: The Model Checker SPIN. IEEE Transactions on Software Engineering, Vol. 23, pp 279-295 (1997)
14 A. Avizienis and L. Chen: On the Implementation of N-Version Programming for Software Fault-Tolerance during Program Execution. In Proc. COMPSAC-77, pp. 149-155, 1977
15 A. Bondavalli, M. Dal Cin, D. Latella and A. Pataricza: High-Level Integrated Design Environment for Dependability (HIDE). Proc. Fifth International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS-99), November 18-20, 1999, Monterey, California, IEEE CS, pp. 87-92 (1999)
16 Innovator version 6.1. MID GmbH, Nuremberg, Germany, http://www.mid.de/en/
17 Object Management Group: XML Metadata Interchange. October, 1998. http://www.omg.org