Code Generation for Hard Real-time Systems from Real-time Statecharts*

Sven Burmester,** Holger Giese, and Wilhelm Schäfer

Software Engineering Group, Department of Computer Science
University Of Paderborn, Germany,
[burmi|hg|wilhelm]@uni-paderborn.de

Abstract. Currently used modeling tools to specify and implement software for embedded systems do not provide adequate means to specify the necessary details to synthesize code automatically from the model definition when hard real-time constraints have to be respected by the implementation. The manual process of testing the hard real-time constraints and adjusting the model or implementation is costly and error-prone. This is also true for UML Statecharts. This paper proposes a syntactic extension and semantic definition of UML Statecharts which provides enough details to synthesize code for hard real-time systems automatically. The algorithm which partitions the model into threads and the corresponding code generation are also described in the paper and show how our approach supports automatic code synthesis.

1 Introduction

There is an increasing demand in industry for sophisticated tools which support the precise specification of software for embedded systems on the one hand and enable automatic code generation on the other hand. A precise specification supports the formal verification of system properties already during the design phase. Code generation avoids many mistakes made when translating a specification manually into code.

Widely used specification methods for those systems include UML Statecharts [1], Statecharts [2], and timed automata [3, 4] in various forms. Their advantage is either their support for powerful modeling concepts which improves usability or their well-defined formal semantics which permits that system properties can be checked automatically based on the formal verification. They do, however, not provide adequate means to specify the necessary details to synthesize code automatically from those models when it comes down to hard real-time constraints.

For the specification of software for embedded real-time systems a number of object-oriented approaches [5–8] including ROOM [5] have been proposed. As most ROOM concepts have been integrated into the UML 2.0 proposal of the main tool vendors [9], they are likely to become soon a part of the standard UML and will therefore

* This work was developed in the course of the Special Research Initiative 614 - Self-optimizing Concepts and Structures in Mechanical Engineering - University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

** Supported by the International Graduate School of Dynamic Intelligent Systems.

be widely available. However, these concepts do not address the temporal behavior of the operational model and therefore do not improve the situation when it comes to the automatic code generation.

Another thread of development is the *UML Profile for Schedulability, Performance, and Time* [10]. The profile defines general resource and time models which are used to describe the real-time specific attributes of the modeling elements such as schedulability parameters or quality of service (QoS) characteristics. Besides an abstract *logic model*, a more concrete *engineering model* can be specified by using these extensions. The engineering model is later used for the required model analysis and code generation. However, it remains an open question in the UML profile how all required details of the engineering model are determined.

Therefore, the current practice when building software for embedded systems, especially with hard real-time constraints is to specify the software on a high abstraction level, then to partition the software to make it run on a real-time operating system in concurrent threads (usually without adequate analysis), to implement it, to test if the time restrictions hold and then usually to re-partition it. Repeating this cycle a number of times is usually very costly but mostly unavoidable.

To close the above identified gap between modeling at high abstraction level and partitioning, the information contained in the specification itself has to be exploited. This paper therefore proposes an approach how UML Statecharts are extended such that they provide enough details to synthesize code which guarantees the specified hard real-time behavior.

The next section gives a detailed example to illustrate the problems of current specification techniques for embedded systems with hard real-time constraints. Section 3 discusses current approaches and their limitations. Section 4 presents our notion of Real-time Statecharts which overcome those limitations and Section 5 sketches the possibilities of some correctness checks on specifications given by this form of Statecharts. Section 6 describes how to synthesize code from such a specification and Section 7 draws a conclusion and sketches current and future work.

2 Example

We use the railroad crossing problem [11] as our running example. When a train crosses the road, gates have to be closed to prevent cars from entering the crossing. Sensors indicate when a train approaches the crossing and leaves the crossing. The reaction to the signals of the sensors is the opening and closure of the gates. Figure 1 depicts the relevant points: A sensor indicates a train approaching the crossing. From this point on the train can still run for a certain distance before the gates start to close. They have to be fully closed when the train passes the gates. When the passing train triggers the `depart` sensor, they can be opened again. The (software) controller of the gates is supposed to be specified by a Statechart.

A first attempt to describe this behavior of the gate controller with a UML Statechart is shown in Figure 2 a). It consists of 3 states. If a train is beyond the section between the two sensors, the controller stays in state `Out_of_Zone`. Entering this section is modeled by the event `approach` which triggers the transition to state `Safe_Zone`.

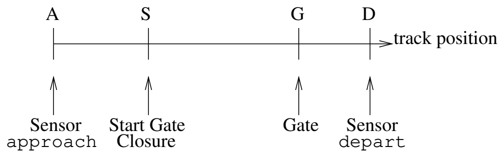

Fig. 1. Topology of the Railroad Crossing

This state is left after t_{safe} time units in order to close the gates by calling action `closeGate()`. t_{safe} is assumed to be the minimum time needed by the train to run from `A` to `S`. The resulting state `Critical_Zone` which represents the situation that the train is just crossing the road, is left, when the train triggers sensor `depart`. The problem, when modeling a behavior like this, is that there is no consideration of time constraints, except the delay after entering the state `Safe_Zone`. As the transitions are assumed to fire within zero-time (which is not implementable in real life systems), the Statechart does not correctly model the described scenario. The gates need some time to close, thus in fact the `closeGate()`-transition has to be fired earlier than specified to prevent that the state `Critical_Zone` will be entered later than t_{safe} time units after entrance of state `Safe_Zone`.

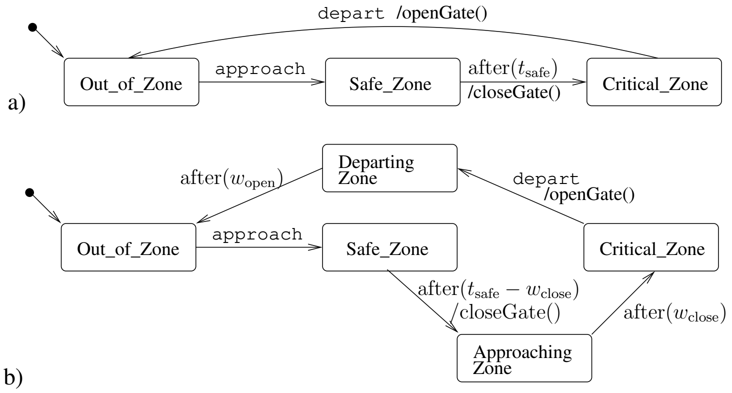

Fig. 2. Approaches for the gate controller

One possibility to model time consumption of actions is the use of the `after`-construct as shown in Figure 2 b), partly adapted from [12]. In order to respect the worst case execution times (WCETs) w_{close} and w_{open} for closing resp. opening the gates, the transition leading to `Approaching_Zone` is fired after $t_{\text{safe}} - w_{\text{close}}$. As in the model the transition fires in zero time, the other two `after`-transitions are needed to model the progress of time.

This description now models correctly that the actions take time, but still consist of transitions that react infinitely fast and execute actions in zero time. If an implementation of the Statechart is executed on a real physical microprocessor, it is of course impossible to react on an event in zero time. If multiple Statecharts/controllers shall execute on a single processor (e.g. for economical reasons), the realization of reaction times near to zero becomes even more problematic. The in the Statecharts employed abstraction of *zero execution time* is thus informally often interpreted to mean *fast enough*. Thus, to guarantee that all tasks gain enough processor load to react in time, deadlines for each required action need to be known. If the Statecharts themselves contain the deadline information, the code generation can respect these restrictions and a scheduling analysis can be performed. Otherwise, we can only test the resulting controller within its context to measure whether all externally defined deadlines are met.

The example illustrates, that UML Statecharts are not sufficient for our demands. In the next section different proposed real-time extensions for Statecharts are introduced and discussed concerning their limitations with respect to model hard real-time constraints and to generate code automatically from the model.

3 Related Work

UML Statecharts offer the `after`- and the `when`-construct to trigger transitions dependant on time. While `after` describes a point in time relative to the entrance of the current source state, `when` denotes an absolute point in time. In order to implement the model of a real-time system the consideration of deadlines and WCETs in the model is required, that is not fulfilled by the mentioned UML constructs (see Section 2).

The CASE tools Rhapsody from iLogix and Rose/RT from Rational support soft real-time system development (cf. [13]). Rhapsody offers code generation for UML Statecharts which even supports *and-states*. Rose/RT combines the concepts of ROOM [5] with UML and therefore supports capsules and ports for a component oriented design, but only supports code generation for a strictly sequential notion of Statecharts related to ROOM charts. In both cases, you have to check whether the synthesized code does not violate the hard real-time requirements after code generation.

Harel [14] introduced two extensions for Statecharts to define time constraints: Using the *Timeout*-construct an event e and a time t is associated with a transition. The transition is triggered t time units after the occurence of the event e. If an action is marked as *scheduled action*, it is not executed during firing a transition, but at a specified later point in time (although the state change happens immediately). In [15] an extension is described, that allows to specify a kind of delay interval of a transition. The bounds of this interval define how much time elapses at least and at most between triggering and firing of the transition. In [16] a similar approach is used. There, the interval describes the points in time, when the source state can be left the earliest and when it has to be left at the latest, relative to the point of entrance. Further, every event has a defined duration, indicating how long it exists and can be consumed after creation. These constructs simplify the modeling of complex behavior and possibly collapse multiple states and transitions into one transition. For example, when modeling the example from Section 2, the state `Safe_Zone` and it's entering and leaving transitions (see Figure 2 b)) are collapsed into one single transition with a scheduled action.

These extensions are useful high-level constructs to specify temporal behavior for systems without any hard real-time constraints, but are not sufficient to specify hard real-time systems. Like the Statechart in Figure 2 b), the models just deliver points in time when transitions are triggered, but do not include sufficient time restrictions or other information, needed for a mapping to an implementation, that guarantees certain execution times.

Also hierarchial extensions for timed automata in form of *hierarchical timed automata* [17] have been developed. They provide most of the powerful modeling concepts of Statecharts. A mapping to multiple parallel running flat timed automata permits to use the model checker UPPAAL [4] to verify the models. Due to missing information about deadlines and WCETs code generation for hard real-time systems as required is not possible.

In [18] the model of an automaton is mapped to a so called *multi-tasking model*. The multi-tasking model describes how the actions of the model are placed in physical threads. The annotation of WCETs and deadlines in the model enables this mapping. As the focus of the work was the implementation of the automaton, no formalization is provided to support verification, but the algorithm that partitions the automaton into threads, guarantees that the deadlines are met. The automaton offers communication via synchronous and asynchronous events. Compared to high-level automata, it is restricted, as it is not possible to use hierarchy, parallelism or to trigger transitions dependant on clocks or variables what makes this model unuseable for our requirements.

4 Real-time Statecharts

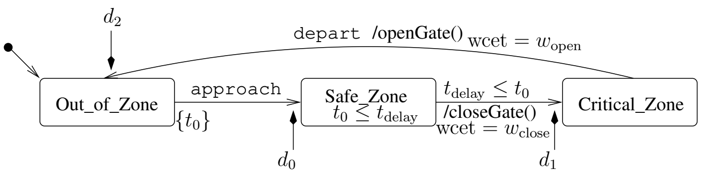

Fig. 3. Real-time Statechart

Both models as presented in Section 2 assume that transitions fire *infinitely fast*, which is unrealistic on real physical devices, especially when considering the execution of the actions which are attached to the transitions. In a realistic model *infinitely fast* has to be replaced by an annotation, describing *how fast* transitions have to be fired or executed. Thus our approach suggests to add deadlines, as for example deadlines d_0, d_1, d_2 are added to the transitions from Figure 2 a) (cf. Figure 3). These time constants specify a relative point in time, which defines the maximum number of time units which can be spent from actually triggering the transaction until the execution of the action has to be finished.

In order to perform timing and schedulability analysis, a WCET is given for each action. Determining these WCETs is usually done by an analysis step as described in

[19]. If the actions are modified during the design process, the WCETs must be re-determined. Note, that deadlines avoid to use extra or help states (as in Figure 2 b)) and thus enable to construct a less complex model in terms of the number of states.

Like in timed automata, our approach also defines clocks that can be reset when a transition fires. As the firing consumes time, these clocks are reset at the point in time when the transition is triggered (e.g. the clock t_0 for the transition from `Out_of_Zone` to `Safe_Zone` in Figure 3).

Using these clocks, time invariants can be specified for a state, defining the point in time, when the state has to be left via a transition. To trigger transitions dependent on a specific point in time, time guards can be specified. Deadlines can either be relative to the triggering of the transition or relative to a clock.

Transitions are triggered when the time guard becomes true, the associated event is available and a guard, consisting of a boolean expression over different variables or methods, is also true. Note that if at least one transition is triggered, one of them is immediately fired, which is sometimes called urgent firing. Non-urgent firing would mean that firing a transition is (unnecessarily) delayed, because the time specifications of the model would still allow a later firing. However, we think that a delay of firing should explicitly be expressed by the corresponding deadlines (and clocks) in the model specification rather than having an underlying semantic definition which actually introduces some sort of non-determinism in the model. To also achieve determinism in case of multiple transitions ready to fire, each transition gets a priority attached, which we omitted in the figures, because there are no conflicts in the presented examples.

The `after`-construct (known from UML Statecharts) can be mapped to time guards and time invariants and thus gets a semantic definition which makes it possible to generate code from this definition. As an example, the `after`-transition from Figure 2 a) is mapped to the time invariant $t_0 \leq t_{\text{delay}}$ in state `Safe_Zone` and the time guard $t_{\text{delay}} \leq t_0$ at its leaving transition. Comparing the models from Section 2 it holds $t_{\text{safe}} = t_{\text{delay}} + d_1$, because `Critical_Zone` is entered at the latest $t_{\text{delay}} + d_1$ after occurence of the `approach` event. Although the use of multiple clocks requires more effort than using the `after`-construct, it has the advantage that the points in time, when transitions are triggered, cannot only be defined relative to the point of entrance of the current state, but also relative to the point of triggering of any previously fired transition or the point of entrance or exit of any previously entered state, because clock-resets can be associated even to the `exit()`- and `entry()`- methods of the states..

To simplify timing analysis (cf. Section 5), the form of the time guards is limited to $\wedge_{t_i \in C}(a_i \leq t_i \leq b_i)$, $a_i \in N$, $b_i \in N \cup \{\infty\}$, where C is the set of clocks. The form of time invariants is limited to $\wedge_{t_i \in C}(t_i \leq T_i)$, $T_i \in N \cup \{\infty\}$. In our experience, this limitation, i.e. the exclusions of arbitrary logic expressions and arithmetic operations on different clock times, does not hamper the modeling of realistic systems and makes it easier for the model developer to build intuitive models rather than very complicated ones.

In summary, the semantic definition of Real-time Statecharts does not have the usual Macrostep and Run-To-Completion semantics of UML Statecharts, because the Zero-Execution time for intermediate steps is not realistic in our application domain. Many actions have significant WCETs. Run-To-Completion semantics would not allow an immediate reaction to any newly raised external event. We define our semantics formally, as given in [20, 21] by a mapping of Real-time Statecharts to a subset of an extended version of hierarchical timed automata as defined in [17]. Such a semantics has already been employed successfully in a similar domain [22] for the un-timed case. In order to still be able to describe the required local synchronization between multiple parallel and-states of a single Real-Time Statechart within a single step, synchronous communication via synchronization-events and -channels, similar to the mechanism described in [4], is also supported.

Apart from the above mentioned extensions, which are partly adapted from timed automata, features from Statecharts, like hierarchy, parallelism and history as well as `entry()`-, `exit()`- and `do()`-operations for states are, of course, provided further on for modeling. To perform temporal analysis, WCETs are also required for these operations. While a specific state is active, its `do()`-operation is executed periodically. The user may specify a time interval for this period. Note also that actions are not limited to integer assignments (like in timed automata), but can be complex method calls in the object-oriented model.

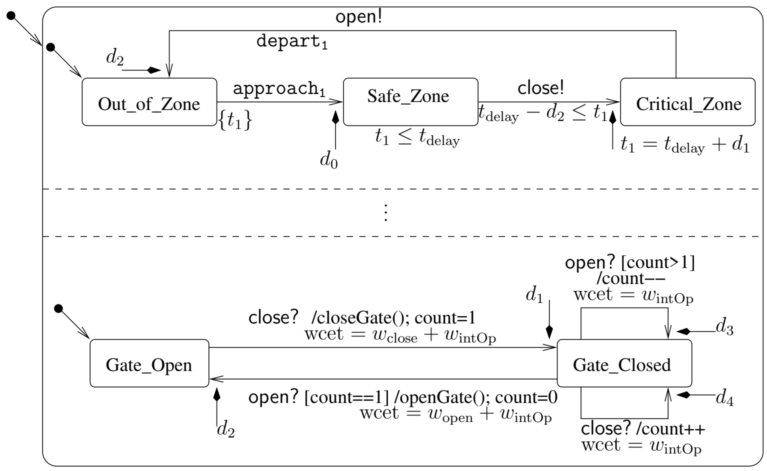

Fig. 4. Parallelism in Real-time Statechart

Figure 4 illustrates a more complex example using Real-time Statecharts including parallelism. The railroad crossing example is extended to multiple tracks. The controller has to ensure that all more or less simultaneously arriving trains have left the crossing, before it opens the gates again. Thus the variable `count` is introduced, keeping track of the current number of trains being inside the section of the controller. For every track i, one branch is created, that handles the events approach$_\mathrm{i}$ and depart$_\mathrm{i}$ resp. and refers to the clock t_i. The WCET for assigning the variable `count` is given by w_{intOp}. To ensure correct values for the variable `count`, the entrance and exit of the states `Critical_Zone` are synchronized with the transitions, which update `count`. The `closeGate()`-transition has to be finished after at most t_{safe} time units after a train approaches. To ensure this requirement closing the gate has to be initiated, taking the maximum delay d_2 into account. Therefore we have to chose the time guard $t_{\mathrm{delay}} - d_2 \leq t_i$.

Real-time Statecharts combine the advantages of Statecharts and of timed automata and extend them by additional annotations. These annotations enable to generate code for real-time platforms on the one hand and offer constructs to model complex temporal behavior on the other hand. The main differences to UML Statecharts are, that they (1) support to model the time consumption of transition execution and (2) have a realistic semantic definition based on timed automata mirroring appropriately the application domain.

5 Temporal Inconsistencies

If temporal inconsistencies in the model definition are excluded, automatic code generation is possible which guarantees the specified real-time behavior. Temporal inconsistencies either mean a so-called *time stopping deadlock* or the contradicting specification of a triple including a particular time guard, time invariant and WCET resp. Our approach allows to detect many of these inconsistencies by static analysis rather than by model checking (symbolic execution), because Real-time Statecharts are always mapped to a restricted form of hierarchical timed automata. We will now illustrate which forms of temporal inconsistencies may arise and which of them are detected in an upfront static analysis of the model.

Using a time invariant to specify the latest point in time, when a state may actually be left by triggering a transition, may cause the specification of unrealistic behavior as defined by contradicting parameter specifications. Consider, for example that the invariant becomes false and no triggered transition exists to leave the state. In this case the time stopping deadlock occurs.

Such deadlocks can be verified for extended hierarchical timed automata by model checking. However, statically the following can be checked: Imagine a state with (a part of) an invariant $t_i \leq T_i$, which is the source of a set of leaving transitions which all have a time guard of the form $T_i + x \leq t_i,\ x > 0$. It is obvious, that once entered, this state will never be left again. The mentioned corresponding transitions are called *dead*, as they will never fire. Time stopping deadlocks will occur in a state if all leaving transitions are dead ones. Dead transitions are detected by static analysis. Only if events or guards are attached to a transition, it is not possible to determine if and when it fires without model checking. Such a transition is called insecure. Thus, static analysis can result in one of three different statements: The Real-time Statechart will cause a time stopping deadlock! , It will not cause a deadlock! or It may cause a deadlock! . Only in the latter case model checking needs to be applied [4] to guarantee that no time stopping deadlock exists.

Other temporal inconsistencies occur if a time guard, a time invariant and a WCET are in contradiction. One case is given by a time guard which can trigger a transition at a point in time, when the execution of the action will not be possible, because the time invariant of the target state may have been exceeded after execution of the action. Consider, for example, a transition with a time guard $t_0 \leq 10$ and an action with a WCET of 4 leading to a state with the invariant $t_0 \leq 12$. If this transition is triggered, for example at $t_0 = 10$, the target state is entered in the worst case at $t_0 = 14$, which violates the time invariant. If such an inconsistency is detected, it can be removed automatically by adjusting the time guard to $t_0 \leq 8$. This adjustment does not add unspecified behavior, because the resulting time guard is just a restricted case of the initial one.

Similarly to the above situation, another inconsistency arises when a time invariant makes it possible to exceed all time guards of all leaving transitions. Consider, for example, a state with the invariant $t_0 \leq 12$ which has 2 outgoing transitions. The time guard of the first one contains $t_0 \leq 10$ and the other one $t_0 \leq 8$. Then the state will never be left after $t_0 = 10$, i.e. a time stopping deadlock does not exist as long as the clock runs and does not exceed 12 time units but is unavoidable thereafter. Without harm, the time invariant can be changed automatically to $t_0 \leq 10$ and no such time stopping deadlock occurs.

Note that automatically adjusting the time specifications as proposed may cause new inconsistencies and results in iterations of the analysis step. However, as each step only decrements time values, they must become negative after enough iterations. This is the time to inform the model developer and let him reconsider his specification, because it must contain the specification of temporal behavior which cannot be implemented.

The sketched static analysis algorithms detect temporal inconsistencies at low costs. Some of the detected inconsistencies are removed automatically. Due to the incompleteness of the analysis, it is a supplement to model checking but cannot, of course, replace it to detect all inconsistencies in the general case.

6 Code Generation

After modeling and checking for time consistency, the Statechart needs to be implemented. The process of developing embedded, real-time systems generally sets a high value on economical factors, so that the number of processors and their power is mostly restricted. Thus our code generation algorithm is designed for single processor systems, whereby also all branches of the and-states need to be mapped to one single processor.

Further one periodic thread is needed ensuring that the Real-time Statechart reacts fast enough to meet all time restrictions. As its period defines *how fast* the Real-time Statechart can react, its determination, that considers the specified attributes (deadlines, etc.) as well as the external determined WCETs, builds the main part of section 6.1. Thus parallelity and the time restrictions lead to a multi-thread application. If multiple Statecharts will be executed on the same processor, schedulability needs to be checked for the set of resulting threads. This analysis, based on the Rate Monotonic Approach [23], is sketched in Section 6.2. After a successfull schedulability analysis code will be generated based on the partitioning results. Section 6.3 gives an example of code generation using Real-Time Java [24] as the target language.

6.1 Partitioning

As mentioned above a Real-time Statechart is mapped to at least one periodic thread, checking for triggered transitions in every period – the so called *main thread*. This thread checks for all transitions which can be triggered from the beginning of its last period until and during the duration of the current period. The checking has to be started that early, because the check in the last period may have just missed a transition which could have been triggered. It was missed because the check happened just before the event occured or its time guard was evaluated to true.

A transition is triggered, if the following four conditions hold: (1) The transition is defined for the current state, (2) the event has occurred in the time interval between the beginning of the last period (of the main thread) and the current point in time (Note that an event, that cannot be consumed immediately, is queued), (3) the time guard is evaluated to true during or after the event happens, (4) the guard is evaluated to true during or after the event happens. The worst case time needed for the whole check (depending on the current state) is denoted by $w_{\mathrm{trig}}(s)$, where s is the current State.

After determining all triggered transitions, their activation times and priorities (cf. Section 4), the first triggered transition with the highest priority is fired. Note that our definition of priorities avoids any nondeterminism in this case. Then clocks are reset and actions are executed.

If the action has such a short WCET, that there is still enough execution time left within the period, it will be executed by the main thread. As it is possible to specify complex actions, their WCETs often do not fit into the main thread. If they are executed within the main thread nevertheless, its execution time will become greater than its period and deadline. Apart from this problem the main thread will not be able to check and – if needed – fire other transitions for the time the action is executed, although this is required in the case of parallelism. Due to these problems, the firing of such transitions is rolled out into a new started aperiodic thread, running in parallel to the main thread. Thus, parallelism is not implemented by multiple concurrent running periodic threads, but by exactly one periodic thread and multiple concurrent running aperiodic threads. Among other things this facilitates the efficient implementation of synchronization.

The still remaining problem is to determine the duration time of the period of the main thread. It needs to be short enough, such that the recognition of triggered transitions happens early enough to guarantee that the actions are executed before their deadlines expire. It should be as long as possible to execute as many transitions as possible within the main thread and thus to minimize resource utilization, because an additional aperiodic thread consumes time and memory. Respecting these conditions, the annotations and restrictions in the Statechart specification as well as the times $w_{\mathrm{trig}}(s)$, w_{start} and w_{end} give limits for the duration time of one period. w_{start} and w_{end} denote the duration for starting and terminating an aperiodic thread. When deriving equations from the specification, several cases need to be distinguished.

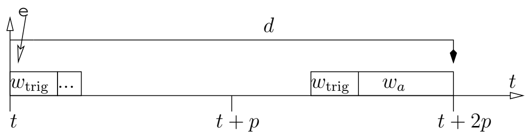

Fig. 5. Determining the period – case 1

Figure 5 shows the first case when $w_{\text{trig}}(s)$ and the to be executed action (WCET is denoted with w_a) fits into the periodic thread. The execution has to guarantee that the action is executed before its deadline expires, i.e. the period is short enough to execute the action before the deadline d. The worst case in this respect, i.e. in terms of a delay between triggering a transition and executing its corresponding action is the following: The main thread begins execution at time t – the beginning of a first period and just misses a transaction which is triggered. That transaction is only checked again and fired at the end of the execution of the next period ($t + p$ until $t + 2p$) such that it just fits into this period (p denotes the duration time of one period). Then $d \geq 2p$ must hold in order to be sure that the action is executed before the deadline expires. Respecting the so called *utilization factor* $\nu \in (0;\ 1]$, defining that a Real-time Statechart shall not gain more than ν percentage of the processor load, obviously $w_{\text{trig}}(s) + w_a \leq p\nu$ must hold for cases where the processor load is shared.

This results in the inequality $p^1_{\text{min}} := (w_{\text{trig}}+w_a)/\nu \leq p \leq \frac{d}{2} =: p^1_{\text{max}}$ determining minimum and maximum values for p in case of executing an action within the main thread (case 1).

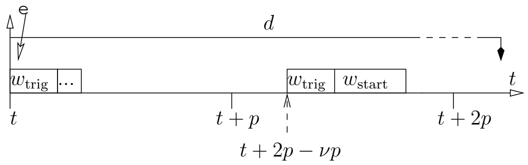

Fig. 6. Determining the period – case 2

A more complex situation occurs when $w_{\text{trig}}(s) + w_a \leq p\nu$ does not hold and the action needs to be rolled out to an aperiodic thread, like shown in Figure 6. Although the start of the aperiodic thread shortens the necessary execution time of the periodic main thread to $w_{\text{trig}}(s) + w_{\text{start}}$, we still compute an upper bound which minimizes rollouts. In any case, the computation time within every period, the main thread gets, is $p\nu$. Even, when this time is not enough to execute the action, the periodic thread is started at least $p\nu$ time units before the end of it's period, cf. Figure 6. In this case, the computation time is not used completely by the periodic thread. The remaining time is already used by the started aperiodic thread. Consider the (trivial) case $\nu = 1$: The delay between triggering the transition and executing the action is given by $p + w_{\text{trig}} + w_{\text{start}}$. Then the action is executed and the aperiodic thread terminates. Thus the delay, the execution and the termination have to fit into the deadline: $d \geq p + w_{\text{trig}} + w_{\text{start}} + w_a + w_{\text{end}} \Leftrightarrow p \leq d - w_{\text{trig}} - w_{\text{start}} - w_a - w_{\text{end}}$.

While the aperiodic thread is executing, the periodic one still runs (with a shorter execution time w'_g) and preempts the aperiodic one once within a period. A detailed analysis (which is given in [20]), respecting these preemptions and $\nu \in (0,\ 1]$ leads to the inequality 1, that uses the substitutions $\alpha = (\nu w_a + \nu w_{\text{end}} + w_{\text{trig}} + w_{\text{start}})/\nu^2$, $\beta = w'_{trig}/\nu$, $\varphi = w_a - w_{\text{trig}} - w_{\text{start}}/\nu$.

$$p \leq d - \alpha - \beta \left\lceil \frac{\nu p - \varphi}{p + \beta} \right\rceil =: p^2_{\max} \quad (1)$$

Considering the necessary execution time $w_{\text{trig}}(s) + w_{\text{start}} \leq \nu p$ leads to another inequality $p^2_{\min} := (w_{\text{trig}}(s) + w_{\text{start}})/\nu \leq p \leq p^2_{\max}$ determining minimum and maximum values for p in case of executing an action, that is rolled out (case 2).

The period has to fit either the first or the second inequality. As a Statechart usually consists of multiple transitions a period is chosen, that fits at least one equation for *every* transition. For the case that a state is entered and a leaving transition becomes triggered immediately, two more inequalities arise, because besides the action the `do()`-operation needs to be executed, too. Further, the period has to fit either the third or the forth equation, too. Analyzing the specified Real-time Statechart leads to a system of inequalities consisting of four times as much inequalities as transitions occur in the Statechart. Thus, choosing the period is a combinatorial problem, that is better solved automatically by a numerical method. If multiple solutions exist, the period for the main thread will be the longest one possible. After determining the period, it is fixed which actions need to be rolled out to aperiodic threads.

6.2 Schedulability

When multiple Real-time Statecharts are executed on the same microprocessor –which results in multiple periodic and aperiodic threads, executed in parallel– the schedulability of the resulting set of threads needs to be checked. A combination of multiple Statecharts would not be schedulable if all of them demand 100% processor load. That is why the utilization factor ν should be defined properly during design. When the employed scheduler and the scheduling algorithm are well-known a scheduling analysis can be performed. In our implementation (cf. Section 6.3) a priority scheduler and the rate monotonic priority assignment algorithm is employed [23].

When performing this analysis the knowledge about the execution order of transitions is used. This order can be derived from the structure of the Real-time Statechart. For example, aperiodic threads, initiated by firing transitions, that are executed sequentially, will never be executed in parallel. All combinations of threads, that can possibly run in parallel are determined and the schedulability of each of these combinations is checked. If one combination is not schedulable the user is informed which combination of transitions leads to schedulability problems and he is required to change the time annotations.

In order to speed up scheduling analysis, we first use a simple method to make a rough estimate and apply a more complex analysis algorithm only if needed [23].

6.3 Implementation

The concepts, introduced in Section 6.1, are of general nature and can be used to realize the implementation in any programming language, supporting concurrency and real-time threads. In this work, Real-Time Java has been chosen.

The two main reasons, why Java cannot be applied to hard real-time systems are that threads cannot be managed by a scheduler and that the garbage collector can always interrupt the current running thread. Both problems have been solved by Real-Time Java, that offers the classes `RealtimeThread` and `NoHeapRealtimeThread`, whose instances can be scheduled by a priority scheduler (if the underlying system provides one). The latter one always executes with a higher priority than the garbage collector. Thus all periodic and aperiodic threads are implemented as `NoHeapRealtime-Threads`.

All objects, needed to start aperiodic threads, are allocated during initialization of the Statechart and managed by a *factory* [25]. As multiple aperiodic threads can be executed and started in parallel, multiple of such instances are created. The management by a factory has the advantage that deallocation is avoided and the time consuming process of allocation has only to be done during initialization.

```
class FMainThread{
  public void executeAperiodicThread(...) {
    FAperiodicStartLogic aperiodicStartLogic = factory.instance(
                                       Factory.F_APERIODIC_STARTLOGIC);
    ScopedMemory scopedMemory = factory.instance(Factory.SCOPED_MEMORY);
    aperiodicStartLogic.setMemoryArea(scopedMemory);
    scopedMemory.enter(aperiodicStartLogic);
    ...
  }
  ...
}

class FAperiodicStartLogic {
  public void run() {
    PriorityParameters pp = factory.instance(Factory.PRIORITY_PARAMETERS);
    pp.setPriority(priority);
    AperiodicParameters ap = factory.instance(Factory.APERIODIC_PARAMETERS);
    ap.getDeadline().set(deadline);
    //create an aperiodic thread in the scopedMemory.
    //FAperiodicThread extends NoHeapRealtimeThread
    FAperiodicThread aperiodic = new FAperiodicThread(pp, ap,
                                        memoryArea, logic);
    aperiodic.start();
    ...
  }
  ...
}
```

Fig. 7. The classes `FMainThread` and `FAperiodicStartLogic`

Another Java-specific problem is that a thread, terminated once, cannot be started again. Thus, these threads cannot be managed by a factory and these instances need to be allocated and deallocated dynamically. In this situation, there is need for deterministic behavior of the garbage collector, which is provided by Real-Time Java, too: Objects allocated in a `ScopedMemory` area, are deallocated iff no object is referenced and no thread is active in this area any more. Thus in our implementation every aperiodic thread is allocated in a scoped memory area, cf. Figure 7.

In lines 3 to 5, reusable instances are taken from the factory, mentioned above. Line 6 just sets a reference. Line 7 causes the execution of the `aperiodicStartLogic` object's `run()`-method in the scoped memory, taken from the factory in line 5. So all allocations, that will take place within this `run()`-method, will be done in the specific scoped memory. In line 20 the allocation of a `FAperiodicThread` object (which extends `NoHeapRealtimeThread`) takes place. The memory area given to the constructor is the scoped memory, set in line 6, so that the allocations within the aperiodic thread (if any) will be done in this scoped memory area, too. Line 22 starts the aperiodic thread.

These described concepts for implementation can be applied to all real-time systems, offering periodic and aperiodic threads (e.g. Real-Time Java). In order to simplify the implementation the concepts should be changed, if the real-time systems follows the *POSIX-sporadic server* policy, which allows (besides the usage of periodic and aperiodic threads) to partition an action into several periods of a single thread [26].

7 Conclusion and Future Work

Real-time Statecharts are a powerful model to specify complex real-time systems provided annotations describing real physical behavior (WCETs, etc.) are given. This enables a mapping of the behavior to real physical threads, which is of general nature and not bound to a specific programming language. Thus an implementation can be realized in any programming language that supports concurrency and real-time threads on a real-time operating system. Right now, the CASE tool Fujaba[1] provides an implementation for Real-Time Java whereas support for C++ and a specific real-time operating system is planned. Further the temporal annotations can be used to perform a simple analysis of temporal consistency. In addition to this analysis our Statechart semantics are mapped to an extended version of hierarchical timed automata. As current work deals with a mapping to the hierarchical timed automata model HUPPAAL, that can be mapped to a flat timed automata model [17], it is planned to use the UPPAAL model checker [4] for verification.

References

1. OMG: Unified Modeling Language Specification Version 1.5. (2002)
2. Harel, D.: STATECHARTS: A Visual Formalism for complex systems. Science of Computer Programming **3** (1987) 231–274
3. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic Model Checking for Real-Time Systems. In: Proc. of IEEE Symposium on Logic in Computer Science. (1992)
4. Larsen, K., Pettersson, P., Yi, W.: UPPAAL in a Nutshell. Springer International Journal of Software Tools for Technology **1** (1997)
5. Selic, B., Gullekson, G., Ward, P.: Real-Time Object-Oriented Modeling. John Wiley & Sons, Inc. (1994)
6. Awad, M., Kuusela, J., Ziegler, J.: Object-Oriented Technology for Real-Time Systems: A Practical Approach Using OMT and Fusion. Prentice Hall (1996)

[1] www.fujaba.de

7. Douglass, B.P.: Real-Time UML: Developing Efficient Objects for Embedded Systems. The Addison-Wesley Object Technology Series. Addison-Wesley (1999) Second Edition.
8. Gomaa, H.: Designing Concurrent, Distributed, and Real-Time Applications with UML. Addison-Wesley (2000)
9. OMG: UML Superstructure Submission V2.0. OMG Document ptc/02-03-02 (2003)
10. OMG: UML Profile for Schedulability, Performance, and Time Specification. OMG Document ptc/02-03-02 (2002)
11. Heitmeyer, C., Jeffords, R., Labaw, B.: Comparing different approaches for Specifying and verifying Real-Time Systems. in Proc. 10th IEEE Workshop on Real-Time Operating Systems and Software (New York) (1993)
12. Lavazza, L., Quaroni, G., Venturelli, M.: Combining UML and Formal Notations for Modelling Real-Time Systems. In: Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE), Vienna Austria. (September 2001)
13. Bichler, L., Radermacher, A., Schürr, A.: Evaluation uml extensions for modeling realtime systems. In: Proc. on the 2002 IEEE Workshop on Object-oriented Realtime-dependable Systems WORDS'02, San Diego, USA, IEEE Computer Society Press (2002) 271–278
14. Harel, D.: Modeling reactive systems with statecharts. McGraw-Hill (1998)
15. Kesten, Y., Pnueli, A.: Timed and Hybrid Statecharts and their Textual Representation. In Vytopil, J., ed.: Formal Techniques in Real-Time and Fault-Tolerant Systems. volume 571 of Lecture Notes in Computer Science, Springer Verlag (1992)
16. Peron, A., A.Maggiolo-Schettini: Transitions as Interrupts: A New Semantics for Timed Statecharts, In Proceedings of TACS '94, LNCS 789, Springer-Verlag (1994)
17. David, A., Möller, M., Yi, W.: Formal Verification of UML Statecharts with Real-Time Extensions. In Kutsche, R.D., Weber, H., eds.: 5th International Conference on Fundamental Approaches to Software Engineering (FASE 2002), April 2002, Grenoble, France. Volume 2306 of LNCS., Springer (2002) 218–232
18. Saksena, M., Karvelas, P., Wang, Y.: Automatic Synthesis of Multi-Tasking Implementations from Real-Time Object-Oriented Models. In: The Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Newport Beach, California (2000)
19. Erpenbach, E.: Compilation, Worst-Case Execution Times and Scheduability Analysis of Statechart Models. Ph.D.-thesis, University of Paderborn, Department of Mathematics and Computer Science (2000)
20. Burmester, S.: Generierung von Java Real-Time Code für zeitbehaftete UML Modelle. Master's thesis, University of Paderborn, Paderborn, Germany (2002)
21. Giese, H., Burmester, S.: Real-Time Statechart Semantics. TechReport tr-ri-03-239, University of Paderborn (2003)
22. Köhler, H., Nickel, U., Niere, J., Zündorf, A.: Integrating UML Diagrams for Production Control Systems. In: Proc. of the 22^{nd} International Conference on Software Engineering (ICSE), Limerick, Irland, ACM Press (2000) 241–251
23. Briand, L.P., Roy, D.M.: Meeting Deadlines in Hard Real-Time Systems: The Rate Monotonic Approach. IEEE Press (1999)
24. Bollella, G., Brosgol, B., Furr, S., Hardin, S., Dibble, P., Gosling, J., Turnbull, M.: The Real-Time Specification for Java^{TM}. Addison-Wesley (2000)
25. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley (1994)
26. Burns, A., Wellings, A.: Real-Timer Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time POSIX. 3rd edn. Addison-Wesley (2001)