Model-Driven Architecture for Hard Real-Time Systems: From Platform Independent Models to Code*

Sven Burmester**, Holger Giese, and Wilhelm Schäfer

Software Engineering Group, University of Paderborn,
Warburger Str. 100, D-33098 Paderborn, Germany
{burmi, hg, wilhelm}@uni-paderborn.de

Abstract. The model-driven software development for hard real-time systems promotes the usage of the platform independent model as major design artifact. It is used to develop the software logic at a high level of abstraction and enables analysis like for example model checking of critical model properties. Ideally, starting with the platform independent model, the platform specific model serves only as an intermediate artifact which is derived automatically, and will finally result in a set of threads whose implementations guarantee the behavior, specified in the platform independent model. However, the current MDA approaches and tools for hard real-time software do not provide this ideal: While some of the MDA approaches could in principle support this vision, most approaches simply do not support an appropriate specification of time constraints in the platform independent model which have to be respected in the platform specific model or in the code. This is also true for UML models and UML State Machines in particular. Our approach overcomes those UML specific limitations by firstly proposing a syntactic extension and semantic definition of UML State Machines which provides enough details to synthesize an appropriate platform specific model that can be mapped to code for hard real-time systems automatically. Secondly, a new partitioning algorithm is outlined, which calculates an appropriate mapping onto a platform specific model by means of real-time threads with their scheduling parameters which can be straight forward transformed to code for the hard real-time system.

1 Introduction

The current practice when building software components with hard real-time constraints is characterized by the following step-wise partially manual process: (1) *Specification:* The software is specified on a high abstraction level (if at all), then (2) *Partitioning:* The software is partitioned into concurrent threads with appropriate periods to make it run on a real-time operating system (usually without adequate analysis), (3) *Implementation:* The software is implemented (often manually, which makes implementation

* 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 founded by the Deutsche Forschungsgemeinschaft.

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

A. Hartman and D. Kreische (Eds.): ECMDA-FA 2005, LNCS 3748, pp. 25–40, 2005.

faults very likely), (4) *Analysis:* It is verified that the software fulfills all real-time constraints in its environment (testing as employed in practice is usually not sufficient for complex software to guarantee the absence of timing errors). If the real-time constraints do not hold, partitioning, implementation and analysis have to be repeated. Repeating this cycle a number of times is usually very costly but often unavoidable.

Consequently, there is an increasing demand to extend model-driven architecture (MDA) [1,2] to design software for embedded hard real-time systems. When using MDA for such systems, the developer would have to specify the so called *Platform Independent Model (PIM)* which describes the system behavior including the real-time constraints which must be met. Ideally, a tool would then automatically partition the specification and map it to the *Platform Specific Model (PSM)*, based on a *Platform Model (PM)* that provides details about the target platform. The PSM describes the active objects and their scheduling parameters which are required to implement the system behavior, specified by the PIM. In the next step, the PSM would be compiled automatically into the platform specific implementation which guarantees a correct implementation of the PIM's semantics. The implementation would guarantee the real-time constraints by construction and thus, no verification of the real-time constraints is required. This would make the above mentioned manual steps *(3) Implementation* and *(4) Analysis* unnecessary. Such guarantees for the derived implementation further permit to analyze other required properties or to reveal faults or inconsistent real-time constraints using the platform independent model rather than considering the much more complex code (e.g. by model checking [3,4,5]).

One reason, why the indicated iterative manual process is followed in practice instead of the MDA approach, is that currently, there exists no support to automatically map a PIM to a PSM that is appropriate for real-time systems. The UML [6] can be considered as the standard to model complex software systems even in the real-time domain [7,8,9,10]. Consequently, we propose in the paper an approach to realize the above outlined vision with UML, even though UML has not been originally designed to support real-time systems and a semantically correct implementation for standard UML State Machines is due to the underlying zero execution time semantics not possible. Although ROOM [11] has finally found its way into UML 2.0, the required support for real-time behavior modeling is still not available, as the ROOM concepts focus on architectural design and do not address the real-time behavior of the operational model at all.

Another thread of development is the *UML Profile for Schedulability, Performance, and Time* [8]. The profile defines general resource and time models which are used to describe the real-time specific attributes of the modeling elements such as scheduling parameters or quality of service (QoS) characteristics. However, it remains an open question in the UML profile how all required details are determined. In a scenario where the developer derives these details from a high-level (platform independent) model and maps them on technical concepts such as threads and periods manually, we still have the problem that this mapping results in an iterative manual process of testing and adjusting the model until the real-time constraints are met. Nevertheless, the profile defines an appropriate level of abstraction to be used as PSM. This PSM can be later used for further model analysis (e.g. scheduling analysis) and code generation.

To provide an appropriate PIM, we first propose a syntactic extension of UML State Machines and a related semantic definition. By enriching the model with deadline information (besides others), our extension provides enough details in the PIM to synthesize a PSM and finally code for hard real-time systems. We provide the PM by a description of worst case execution times (WCETs) of local side-effects and of the code fragments that will be used in the automatically generated implementation of the state machine.

For the automatic derivation of the PSM from our extended notion of State Machines, we developed an algorithm for automatic partitioning and for automatic derivation of scheduling parameters. The algorithm takes CPU time sharing on a single micro processor into account. An automatic implementation usually leads to less faults than a manual implementation. The automatic partitioning respects the deadlines from the PIM and the WCETs from the PM.

Therefore, the algorithm for automatic *Partitioning* and *Implementation* guarantees that all real-time requirements are met, which makes the *Analysis* unnecessary and avoids the costly iterative process of *Partitioning*, *Implementation* and *Analysis*. If the algorithm fails to provide a partitioning, the model is not realizable.

The next section presents our approach for platform independent modeling of hard real-time systems and relates it to standard UML models. Section 3 describes the platform model and the component's deployment. Section 4 shows in detail how to derive a platform specific model and finally code. Section 5 discusses current approaches of specification techniques for embedded systems with hard real-time constraints and their limitations. Finally, Section 6 draws a conclusion and sketches current and future work.

2 Platform Independent Models

In this section, we first describe how to specify the system's structure. Then, we discuss in detail how to specify the behavior of components of embedded real-time systems with UML and with our approach. Finally, we present our analysis methods.

2.1 Structure Modeling

Embedded real-time systems consist of a complex architecture of components (cf. Figure 5). In [3], we have presented an approach how to specify the architecture and complex real-time communication between the components by UML component diagrams and patterns respectively. Our approach further permits to verify the component's interconnection by means of compositional model checking assuming that each single component behaves as specified. How the single component's real-time behavior is specified and how it is correct implemented automatically is described in the remainder of this paper.

2.2 Behavior Modeling

We use an example from the RailCab research project[1] as our running example. The vision of the RailCab project is a rail system where autonomous operating shuttles apply

[1] http://www-nbp.upb.de/en/index.html

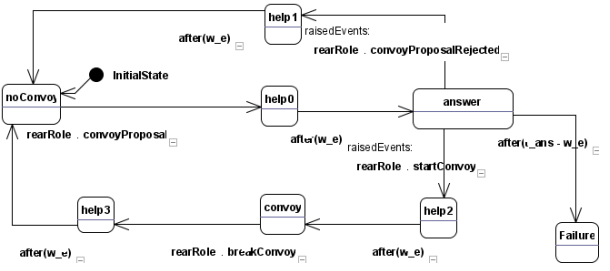

Fig. 1. UML approach to model the shuttle coordination

the linear drive technology used in the Transrapid, but travel on the existing passive track system of the standard railway. One particular problem, which has been previously described in [3], is to reduce the energy consumption due to air resistance by forming convoys whenever possible. Such convoys are created on-demand and require small distances between the shuttles in order to achieve significant economies.

Building convoys changes the shuttles' behavior (e.g. the way of accelerating and braking). Thus, It must be guaranteed that all involved shuttles of a convoy switch to convoy mode in an appropriate and predictable amount of time which results in a number of hard real-time constraints.

After receiving a convoyProposal message, that denotes a request to build a convoy, we demand for the communication that the shuttle answers within the time t_{ans} with rejection (message convoyProposalRejected) or with acceptance (startConvoy).

In a first attempt to describe this coordination with a UML State Machine, the state machine would switch to an intermediate state when receiving convoyProposal. This intermediate state would be left via a transition labeled with after(t_{ans}) to switch to a Failure state if no answer was sent during this time.

The semantics of such a model assumes the transitions to be fired within zero-time, but this is not realizable in an implementation in real life systems due to three reasons: (i) Consuming or raising events or executing side-effects consumes time. (ii) An implementation of a state machine requires a task which periodically checks if transitions are triggered. As only positive, *non-zero* periods are realizable, this leads to a further delay. (iii) If other processes are executed on the processor, further delays occur due to scheduling.

One possibility to model time consumption of raising events or executing side-effects is the use of the after-construct as shown in Figure 1. In order to respect the worst case execution time w_e for consuming or raising events the help-states are introduced. They are entered when an event is consumed or fired and left after w_e (to simplify the example we assume that consuming and raising of events consumes the same amount of time).[2]

Such a description models correctly that the actions consume time (cf. (i) above), but still consist of transitions that react infinitely fast (cf. (ii)) and do not respect scheduling

[2] Note when regarding Figure 1 that we denote the sending of a message msg to target tgt by tgt.msg. Receiving from receiver rcv is denoted by rcv.msg.

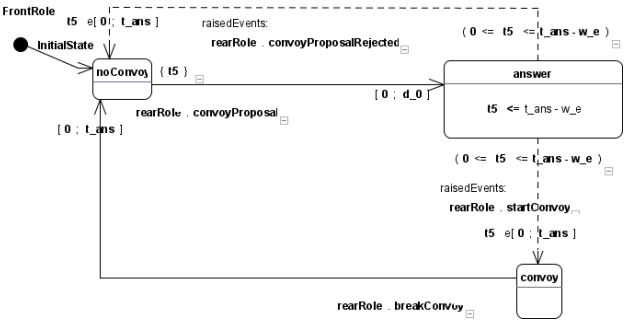

Fig. 2. Real-Time Statechart

delays (cf. (iii)). Further, the after-construct is used in 2 different ways: after($t_{ans} - w_e$) specifies the point in time when the according transition has to fire (as proposed by the UML). Contrary to this, after(w_e) is used to model the progress of time while raising an event.

The example illustrates, that UML State Machines are not practical for our demands and that there is need for a realistic model that supports the specification of hard real-time constraints like WCETs and upper bounds for reaction times.

The abstraction of *zero execution time*, employed in UML State Machines, is often interpreted to mean *fast enough*. Thus, to specify *how fast* they have to react, we propose to specify deadlines for each required side-effect. Thus, in our *Real-Time Statechart* model [12,13], which is an extension of the UML State Machine model, transitions are not assumed to fire *infinitely fast*, which is unrealistic on real physical devices (especially when considering the execution of the actions attached to the transitions), but it is possible to specify deadlines for each transition which in turn determine what *fast enough* really is.

These time constants specify a relative point in time defining the minimum time (always 0 in this example, see Figure 2) and the maximum time (d_0, t_{ans}) until the firing of the transition has to be finished. These points in time are either absolute in relation to the point in time when the transition has been triggered (e.g. the transition from noConvoy to answer) or relative to a clock. In the example, the deadline $t_5 \in [0; t_{ans}]$ of the transition from answer to convoy is relative to the clock t_5. t_5 is reset to zero when switching to state answer (indicated by $\{t_5\}$ similar to the notion in timed automata [14,15]). The clock is reset at the point in time when the transition is triggered. The deadlines avoid to use extra or help states (as in Figure 1) and thus enable to construct a less complex model in terms of the number of states.

Further, we enhance the model –similar to timed automata– by time invariants defining the point in time when the state has to be left via a transition. The state answer is only valid as long $t_5 \leq t_{ans} - w_e$ holds. To trigger transitions dependent on a specific point in time, time guards are specified (e.g. $0 \leq t_5 \leq t_{ans} - w_e$).

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. We distinguish between *urgent* transitions (visualized by solid ar-

rows) firing immediately when they are triggered and *non-urgent* transitions (visualized by dashed arrows). The latter ones may be delayed when the time specifications of the model still allow a later firing [14]. Urgent transitions are similar to eager transitions in [16] and non-urgent transitions are similar to delayable or lazy transitions in [16]. They are used to model different possible alternatives in the communication protocol. This introduced non-determinism is resolved in Figure 3 showing the whole shuttle behavior.

The after-construct is mapped to a time guard and a time invariant and thus gets a semantic definition which makes it possible to generate code from this definition. 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.

The form of the time guards is limited to $\wedge_{t_i \in C}(a_i \leq t_i \leq b_i)$, $a_i \in \mathbb{N}$, $b_i \in \mathbb{N} \cup \{\infty\}$, where C is the set of clocks. The form of time invariants is limited to $\wedge_{t_i \in C}(t_i \leq T_i)$, $T_i \in \mathbb{N} \cup \{\infty\}$. 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.

The semantic definition of Real-Time Statecharts does not have the usual macrostep and run-to-completion semantics of UML State Machines, 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 [13] 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 [18] for the un-timed case. In order to still be able to describe the required local synchronization between multiple orthogonal states of a single Real-Time Statechart within a single step, synchronous communication via synchronization-events and -channels, similar to the mechanism described in [14], is also supported.

Apart from the above mentioned extensions, which are partly adapted from timed automata, features from UML State Machines like hierarchy, parallelism and history as well as entry()-, exit()- and do()-operations for states are, of course, provided further on. While a specific state is active, its do()-operation is executed periodically. The user may specify a time interval for this period. Actions are not limited to integer assignments (like in timed automata), but can be complex method calls in the object-oriented model. The WCETs are respected in the PM (see Section 3).

Figure 3 shows the whole shuttle behavior, consisting of three orthogonal states.[3] The upper orthogonal state realizes the described part of the communication protocol. The lower orthogonal state realizes the opposite part. The orthogonal state in the middle

[3] Note that in our CASE tool *Fujaba* (www.fujaba.de) the dashed lines between orthogonal states are not visualized.

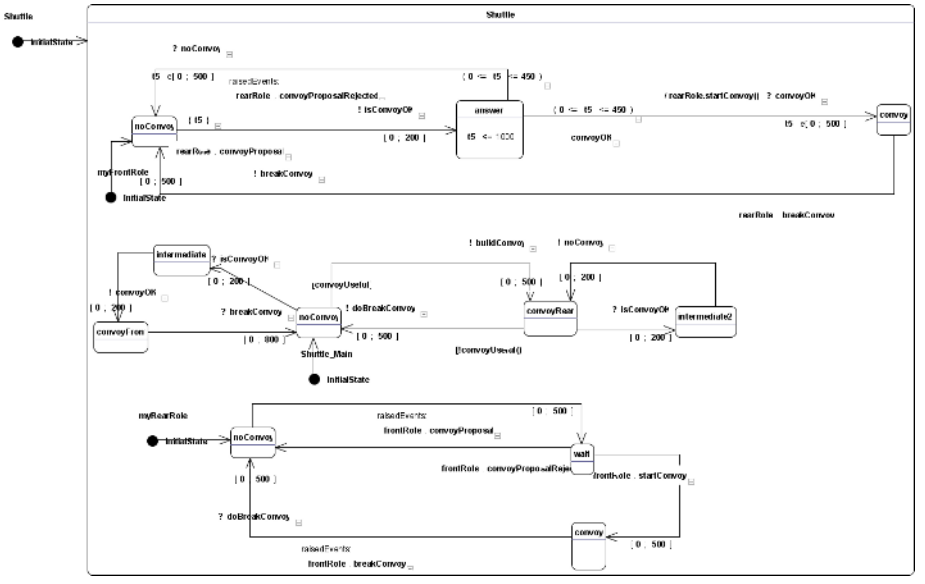

Fig. 3. Behavior of a shuttle component

synchronizes both roles. It initiates the building and the breaking of the convoy. In this simplified example, convoys consisting of maximal two shuttles are build.

Real-Time Statecharts combine the advantages of UML State Machines and of timed automata and extend them by additional annotations. These annotations enable to generate the PSM and finally 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 State Machines 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.

2.3 Model Analysis

Generating a PSM, consisting of active objects and deadlines, that guarantee the real-time constraints as specified in the model is of course only possible, when the model does not contain any conflicts between the declarative elements such as time guards and time invariants. A possible conflict is for example when multiple real-time constraints are contradicting and thus no behavior exists which fulfills them (time-stopping deadlock).

To exclude such conflicts, the full state space of a Real-Time Statechart model has to be checked in the general case. Due to the well-defined semantics of Real-Time Statecharts [13], which map their behavior to hierarchical timed automata as employed in the model checker UPPAAL [19,14], we first map them to hierarchical timed automata and then feed them into the vanilla extension of UPPAAL [19] which flattens them

in an additional preprocessing step. Then, this flat timed automata model is checked with UPPAAL for the absence of time-stopping deadlocks or other required properties expressed with a restricted temporal logic.

When a time-stopping deadlock has been found, we have to conclude that the final state of the delivered error trace contains a conflict. Pinpointing the root source of the problem is a complex problem which remains to be done manually.

While model checking the PIM provides a high cost solution in the general case, we can do much better for specific failure classes where the complex dependencies which result from the synchronization between orthogonal states are ignored when the deployment and thus the platform model is known.

Imagine, as one example for such a static analysis, 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 and a time-stopping deadlocks occurs.

Our additional static analysis algorithms employed upfront detect such temporal inconsistencies at low costs. 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. The pessimistic analysis further indicates whether model checking is required at all or whether the much simpler static checking for temporal inconsistencies has been sufficient.

3 Platform Models and Deployment

In order to generate the PSM, WCETs are required for all actions (side-effects, entry(), exit(), and do()- operations) and for the elementary instructions that build the code fragments realizing the Real-Time Statechart behavior (e.g. checking guards, raising events, etc.).

3.1 Deployment

As the WCETs are platform-dependent, we first deploy our components (whose behavior is each specified by a Real-Time Statechart) by a UML deployment diagram. In such a deployment diagram, we assign the component instances of our systems to dedicated nodes and the cross node links to available network connections in form of busses or direct communication links. Given such an assignment, we can further look into the specific characteristics of the different nodes as described in the platform model.

3.2 Platform Models

In the platform model, the relevant characteristics such as CPU type, operating system, etc. are described. Therefore, available techniques to determine these single WCET values as described in [20] can be employed. They allow to annotate these values to the platform specific view of the behavioral elements such as methods and elementary instructions. The WCETs of the code fragments of a Real-Time Statechart can then be determined by summing up the execution times of the elementary instructions and more complex methods.

3.3 Model Analysis

To analyze the resulting model with platform specific annotations, we extend our timed automata model for model checking as well as our static analysis technique such that it also reflects the WCET behavior of the side effects of the transitions.

A temporal inconsistency can, for example, 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.

Such problems can be detected using model checking. In addition our static analysis algorithms can be upfront detect some of these temporal inconsistencies at low costs as in the case of the PIM analysis.

4 Synthesizing Platform Specific Models and Code

After modeling and analyzing the PIM with components and Real-Time Statecharts and specifying the platform specific WCET information in the PM and the deployment, we have to map the components and links to active objects and to network and communication links to come up with the final platform specific model. In our case the PSM can be described by the UML Profile for Schedulability, Performance, and Time [8], as it allows the specification of priorities, periods, and deadlines for active objects. We use it as platform *specific* model, as these values, which we derive automatically from the platform *independent* model, are different for different platforms.

When building real-time systems, cost saving requires to minimize hardware costs. Consequently, the number of processors and their power is restricted. Thus our mapping algorithm is designed for single processor systems, whereby all branches of the orthogonal states are mapped to one single processor. In case the system consists of multiple components, deployed on different processors, every component executes on exactly one processor. Thus, the mapping algorithm can then be applied, too.

One periodic thread ensures that the Real-Time Statechart reacts *fast enough* to meet all time restrictions. The thread's period defines *how fast* the Real-Time Statechart reacts. Its determination, that considers the specified attributes (deadlines, etc.) as well as the externally determined WCETs, builds the main part of Section 4.1.

As every Real-Time Statechart is implemented as exactly one active object which will be implemented as periodic thread (and possibly multiple aperiodic threads), the number of concurrently running threads can become large when plenty of Real-Time Statecharts are executed on the same processor. If this is the case (e.g., for UML models with a large number of active objects or components), we propose to combine multiple Real-Time Statecharts into a single one using orthogonal states to optimize the result of the partitioning. Using such a grouping, an unacceptable overhead due to a large number of threads is avoided and we still resolve the partitioning and scheduling problem by employing the proposed code generation algorithm.

4.1 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 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 occurred 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 and the points in time when they became activated, the first triggered transition is fired. Then clocks are reset and actions are executed.

If the action has such a short WCET, such 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 would become greater than its period and deadline. Apart from this problem, the main thread would 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 orthogonal states. Due to these problems, the firing of such transitions is rolled out into a new started aperiodic thread, running concurrently to the main thread. Thus, orthogonal states are 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 main thread's period. On the one hand, 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. On the other hand, 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. We determine the period for a target system, scheduled by a priority scheduler [21]. When deriving equations to determine the period from the specification, several cases need to be distinguished.

Figure 4a shows the first case when $w_{\mathrm{trig}}(s)$ and the action to be executed (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

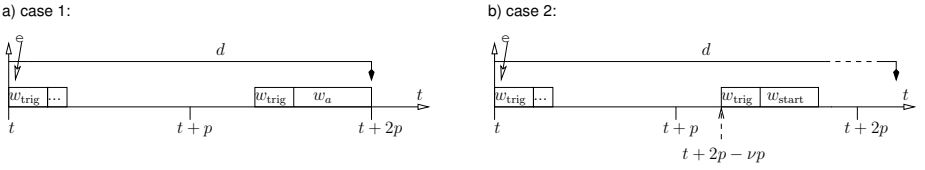

Fig. 4. Determining the period

the action before the deadline d. The worst case 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 transition which is triggered. As we apply priority scheduling, that transition 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 $p^1_{\min} := (w_{\text{trig}}(s) + w_a)/\nu \leq p \leq \frac{d}{2} =: p^1_{\max}$ determining minimum and maximum values for p in case of executing an action within the main thread (case 1).

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 4b. 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 4b. 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 [22]), respecting these preemptions and $\nu \in (0,\ 1]$ leads to the inequality 1, that uses the substitutions $\alpha = (\nu w_a + \nu w_{\text{end}} + w_{\text{trig}} + w_{\text{start}})/\nu^2$, $\beta = w'_{trig}/\nu$, $\varphi = w_a - w_{\text{trig}} - w_{\text{start}}/\nu$.

$$p \leq d - \alpha - \beta \left\lceil \frac{\nu p - \varphi}{p + \beta} \right\rceil \tag{1}$$

Applying a numerical algorithm leads to the solutions in the form $p \leq p^2_{\max}$. 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 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.

4.2 Platform Specific Model

Figure 5 depicts the structural view of the PIM and the according generated PSM. The Shuttle component is transformed to the active class MainThread, realizing the periodic main thread and to the class Shuttle, realizing the logic of the component. The determined period, which is equal to its deadline, is annotated as proposed by the Profile for Performance, Schedulability, and Time. The priority is determined according to the deadline monotonic approach [21]: The thread with the shortest deadline, achieves the highest priority. Note that the deadlines of the aperiodic threads, that execute long side-effects are specified in the PIM.

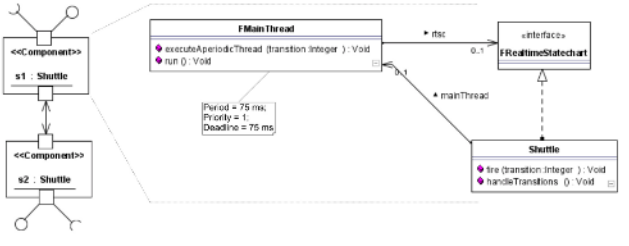

Fig. 5. PIM and PSM of the shuttle system

4.3 Model Analysis

When implementing applications for embedded real-time systems, resource restrictions need to be taken into account. Memory and computation time are usually the restricted resources in embedded systems. As the structure of our models is static and thus there is no need for dynamic instantiation, the required memory can be derived straight forward and is fixed after partitioning. To check if sufficient computational power is available, especially when multiple Real-Time Statecharts or other processes have to be executed on one microprocessor, scheduling analysis is performed. Note that even when the sum of all processes' utilization factors is less or equal 100%, schedulability cannot be guaranteed without adequate analysis [21].

In order to speed up scheduling analysis, we first use Liu and Layland test [23] to make a rough estimate and apply Lehoczky's, Sha's, and Ding's analysis algorithm [24] only if needed. If the set of *all* threads is not schedulable, we exploit the knowledge about the possible concurrently executed threads which 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 concurrently. All combinations of threads, that can possibly run concurrently are determined and it is sufficient to check the schedulability for all these combinations.

4.4 Code Generation

Using the automatically generated PSM, the mapping to a real-time target platform, supporting priority scheduling, is straight-forward. Currently, we support the generation of Real-Time Java [25] and C++ for an appropriate real-time operating system. In this generation step, active objects are mapped to real-time threads. The result of this mapping can be imported into our CASE Tool Fujaba by its reengineering capabilities.

5 Related Work

Currently available approaches for the specification and implementation of hard real-time systems have the following disadvantages: Either, they offer the required higher level modeling concepts, but provide no partitioning and code generation concepts which ensure the specified hard real-time behavior of the model, or they support code generation which guarantees timing behavior, but are already platform specific models.

In [9], Rational Rose models are extended with information needed for scheduling and partitioning in form of periods and action WCETs. This information is then used to distribute the components automatically to multiple processors and to guarantee schedulability. This approach is, however, rather limited as synchronization within the components (usually described by Statecharts) is not supported.

Hierarchical timed automata [17], which are a hierarchial extensions of timed automata [14,15], provide most of the powerful modeling concepts of Statecharts. A mapping to multiple parallel running flat timed automata permits to verify the model by using the model checker UPPAAL [14]. In [26], locations of a flat UPPAAL automaton are associated with tasks inclusive WCETs and deadlines. This extension enriches the model with the information required for code generation and a prototype synthesizing C-Code has been implemented. As the code generation approach is restricted to flat automata, it does not take the additional syntactical constructs of hierarchical timed automata into account. The code generation scheme is not really sufficient for hard real-time systems, as it does not take into account the delays that occur when transitions are fired, arguing that these delays are small compared with the WCETs.

Modecharts [27] are another high-level form of state transition systems for the specification of real-time systems. Actions are executed only while residing in states and not when firing transitions. The model respects that actions require time and thus they are associated with deadlines or –if needed– with periods. Timing constraints like deadlines and trigger conditions are specified just relative to the current state's (mode's)

point of entry and not relative to preceding states. [28] describes code generation for the target language ESTEREL, but the generated implementation regards only the timing intervals, triggering the transitions and not the deadlines or periods.

SAE AADL (Society for Automotive Engineers Architecture Analysis & Design Language) [29], successor of MetaH,[4] specifies a system on the PSM level. A SAE AADL model consists of multiple Threads, annotated by a priority and a frequency and can therefore be mapped to code automatically. Tool support for modeling is currently restricted to text based editors.

The application framework VERTAF [30] and the automata model presented in [31] specify the required real-time constraints and thus enable an automatic implementation. These approaches are not applicable for complex systems, as their models are rather restricted: [31] applies just a flat automata model, [30] specifies active object on the implementation level.

Currently available CASE tools Rhapsody, Rational Rose/RT, Statemate, TelelogicTau, and Artisan Real-time Studio Professional for UML State Machines can only generate code from the logical behavior, while an appropriate mapping onto threads and scheduling parameters in form of the synthesis of a platform specific model remains to be determined in a manual process. To the best of our knowledge all existing UML CASE tools also fail to close the gap between high level models and the automatic implementation of hard real-time systems.[5] In contrast, the presented approach supports the automatic synthesis of the PSM from a given PIM and PM.

6 Conclusion and Future Work

Our approach, consisting of components and Real-Time Statecharts, permits to specify complex real-time systems following UML notations and the MDA approach at the PIM level. This platform independent description can then be mapped automatically to a platform specific model, provided that a target platform description in form of annotations describing real physical behavior (WCETs) are given. The PSM describes real-time threads, which are of general nature and not bound to a specific programming language or RTOS environment. Thus, an implementation can be realized in any programming language that provides real-time priority scheduling. Different analysis methods are applied on the different levels to achieve correct models.

Right now, the open-source UML CASE tool Fujaba supports modeling with UML components and Real-Time Statecharts including model checking and code generation for Real-Time Java and C++ from UML components and Real-Time Statecharts. We are currently extending Fujaba to explicit visualize the generated PSMs and to permit manual adjustments like adding other threads to the system's nodes. In this context, we prove if the standard UML Profile for Schedulability, Performance, and Time

[4] www.htc.honeywell.com/metah

[5] We refer to [7] for a judgment that Rhapsody (www.ilogix.com) and Rose/RT (http://www-306.ibm.com/software/rational/) only support soft real-time system development. We further evaluated Artisan Real-time Studio Professional (www.artisansw.com), Statemate (www.ilogix.com), Rational Rose/RT, and Telelogic Tau G2 Developer (www.telelogic.com) on our own.

is sufficient or if extensions like for example the HIDOORS Profile [32,33] are required. Automatic grouping of Real-Time Statecharts and modular code generation for deployment-time grouping is planned future work.

References

1. Allen, P., ed.: The OMG's Model Driven Architecture. Volume XII of Component Development Strategies, The Monthly Newsletter from the Cutter Information Corp. on Managing and Developing Component-Based Systems. (2002)
2. Object Management Group: MDA Guide Version 1.0. (2003) Document omg/2003-05-01.
3. Giese, H., Tichy, M., Burmester, S., Schäfer, W., Flake, S.: Towards the Compositional Verification of Real-Time UML Designs. In: Proc. of the European Software Engineering Conference (ESEC), Helsinki, Finland, ACM Press (2003)
4. Burmester, S., Giese, H., Hirsch, M., Schilling, D.: Incremental Design and Formal Verification with UML/RT in the FUJABA Real-Time Tool Suite. In: Proceedings of the International Workshop on Specification and vaildation of UML models for Real Time and embedded Systems, SVERTS2004, Satellite Event of the 7th International Conference on the Unified Modeling Language, UML2004. (2004)
5. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (2000)
6. Object Management Group: UML 2.0 Superstructure Specification. (2004) Document: ptc/04-10-02 (convenience document).
7. Bichler, L., Radermacher, A., Schrr, 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
8. Object Management Group: UML Profile for Schedulability, Performance, and Time Specification. OMG Document ptc/02-03-02 (2002)
9. Gu, Z., Kodase, S., Wang, S., Shin, K.G.: A Model-Based Approach to System-Level Dependency and Real-Time Analysis of Embedded Software. In: The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, Toronto, Canada. (2003)
10. Masse, J., Kim, S., Hong, S.: Tool Set Implementation for Scenario-based Multithreading of UML-RT Models and Experimental Validation. In: The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, Toronto, Canada. (2003)
11. Selic, B., Gullekson, G., Ward, P.: Real-Time Object-Oriented Modeling. John Wiley & Sons, Inc. (1994)
12. Burmester, S., Giese, H., Tichy, M.: Model-Driven Development of Reconfigurable Mechatronic Systems with Mechatronic UML. In Assmann, U., Rensink, A., Aksit, M., eds.: Model Driven Architecture: Foundations and Applications. Volume 3599 of Lecture Notes in Computer Science (LNCS)., Springer Verlag (2005) 47–61
13. Giese, H., Burmester, S.: Real-Time Statechart Semantics. TechReport tr-ri-03-239, University of Paderborn (2003)
14. Larsen, K., Pettersson, P., Yi, W.: UPPAAL in a Nutshell. Springer International Journal of Software Tools for Technology **1** (1997)
15. 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)
16. Bornot, S., Sifakis, J., Tripakis, S.: Modeling Urgency in Timed Systems. In Roever, W.P.d., Langmaack, H., Pnueli, A., eds.: Compositionality: The Significant Difference; COMPOS '97, Bad Malente, Germany, September 8 - 12, 1997. Volume 1536 of Lecture Notes in Computer Science., Springer Verlag (1998) 103–129

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. 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
19. David, A., Moeller, M.: From HUPPAAL to UPPAAL: A translation from hierarchical timed automata to flat timed automata. In: TechReport BRICS RS-01-11, Department of Computer Science, University of Aarhus. (2001)
20. 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)
21. Buttazzo, G.C.: Hard Real Time Computing Systems: Predictable Scheduling Algorithms and Applications. Kluwer international series in engineering and computer science : Real-time systems. Kluwer Academic Publishers (1997)
22. Burmester, S.: Generierung von Java Real-Time Code fr zeitbehaftete UML Modelle. Master's thesis, University of Paderborn, Paderborn, Germany (2002)
23. Liu, C.L., Layland, J.W.: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. Journal of the ACM **20** (1973)
24. Lehoczky, J., Sha, L., Ding, Y.: The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In: Proceedings of the 10th Real-Time Systems Symposium. (1989)
25. Bollella, G., Brosgol, B., Furr, S., Hardin, S., Dibble, P., Gosling, J., Turnbull, M.: The Real-Time Specification for Java^{TM}. Addison-Wesley (2000)
26. Amnell, T., David, A., Fersman, E., Pettersson, M.O.M.P., Yi, W.: Tools for Real-Time UML: Formal Verification and Code Synthesis. In: Workshop on Specification, Implementation and Validation of Object-oriented Embedded Systems (SIVOES'2001). (2001)
27. Jahanian, F., Mok, A.: Modechart: A Specification Language for Real-Time Systems. In: IEEE Transactions on Software Engineering, Vol. 20. (1994)
28. Puchol, C., Mok, A., Stuart, D.: Compiling Modechart Specifications. In: 16th IEEE Real-Time Systems Symposium (RTSS '95), Pisa, Italy. (1995)
29. Feiler, P.H., Gluch, D.P., Hudak, J.J., Lewis, B.A.: Embedded Systems Architecture Analysis Using SAE AADL. Technical Report CMU/SEI-2004-TN-005, Carnegie Mellon University (2004)
30. Hsiung, P.A., Su, F.S., Gao, C.H., Cheng, S.Y., Chang, Y.M.: Verifiable Embedded Real-Time Application Framework. In: Seventh Real-Time Technology and Applications Symposium (RTAS '01), Taipei, Taiwan. (2001)
31. 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)
32. Richard-Foy, M., Hunt, J.J.: The HIDOORS Profile: Applying the Scheduling, Performance and Time Profile to Realtime Java Development. In Amann, U., ed.: Proc. of Model Driven Architecture: Foundations and Applications (MDAFA 2004), Linköping, Sweden. (2004)
33. Meunier, J.N., Lippert, F., Jadhav, R., Harding, N.: MDA and Real-Time Java: The HIDOORS project. In Akehurst, D., ed.: Proc. of Second European Workshop on Model Driven Architecture (MDA) with an emphasis on Methodologies and Transformations (EWMDA-2 2004), Canterbury, England. (2004) 89–95