A Unified Model for Analysis of Real-Time Properties*

Oana Florescu, Jeroen Voeten, and Henk Corporaal

Information and Communication Systems Group, Faculty of Electrical Engineering
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
o.florescu@tue.nl

Abstract. A model-driven design approach that provides a unified way of modeling real-time systems, covering both functional and timing characteristics, enables designers to reason about different properties. Refinements of the model should be easy to construct in order to obtain a complete specification from which the implementation may be automatically generated.
Previous work has shown how, in the Software/Hardware Engineering design method, the preservation of all system properties when synthesizing the model can be guaranteed upto to a small time-deviation. This technique is well suited for control-systems, in which execution times of actions are very small, so the time-deviations obtained are small as well. However, it is not applicable for systems containing time-intensive computations because the time-deviations become large. This paper proposes an approach for imposing less restrictive timing constraints in order to decrease the time-deviation between model and implementation for this kind of systems. Furthermore, we present a mechanism for analyzing their timing behavior together with the estimation of the time-deviation.

1 Introduction

The main purpose of engineering models is to help engineers understand the interesting aspects of a system, before getting to the expense and trouble of actually building it. Traditional forms of engineering have a well-developed modeling methodology and their use of models is generally recognized as a useful and effective technique. But software engineering, and particularly real-time embedded software, is still a rather emerging discipline, which is applied to increasingly complex systems. Although its modeling techniques are often unreliable [1], software models have a unique and remarkable advantage over most of other engineering models: they can be used to automatically generate executable programs for particular platforms. Starting with a simplified and highly abstract model, refinements can be carried on until a complete specification is obtained, including

* This work is being carried out as part of the Boderc project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Senter TS program.

all the details necessary in the final product, and from which adequate computer tools can generate an implementation. The initiative of the Object Management Group (OMG) for Model-Driven Architecture (MDA) [2] shows that interest in technologies for supporting model-driven development has increased.

Embedded systems nowadays, found in cars, airplanes, medical devices, employ real-time software components to synchronize and coordinate different processes and activities. Their behavior must meet hard timing constraints, either for people's safety or simply to make things work correctly. The correctness of a hard real-time software component, that works together with other software and hardware components to achieve the specified behavior, depends not only on the logical result obtained but also on the moment in time the result was ready. For predictably designing such systems, an appropriate methodology should provide [3]: (i) a suitable modeling technique, which can appropriately capture functional and timing properties in models, and (ii) a mechanism for generating the implementation from the model while preserving the properties verified.

Traditional scheduling analysis like Rate-Monotonic Analysis (RMA) [4] allows engineers to reason about schedulability of tasks (software components) on the target platform. RMA has grown to a collection of quantitative methods and algorithms that can specify, analyze and predict timing behavior of real-time software systems. Nevertheless, a RMA model does not capture aspects of the functional behavior, so refinements towards a synthesizable description of a system cannot be made. On the other hand, models built using formal techniques like process algebras (Concurrent Communicating Systems - CCS [5] or Communicating Sequential Processes - CSP [6]), or timed automata [7] allow reasoning about both functionality and timing, but they are not used for software synthesis.

A unified executable model, upon which both timing and functional analysis can be performed, helps designers in reasoning about all kinds of aspects of a system in a unified way. Moreover, in the MDA context this model should be easily refinable towards a complete system specification from which, through a property-preserving transformation mechanism, the realization of the system can be obtained. Although there is some ongoing research in this direction, it is still an immature area. This paper is an attempt to address this issue and to go a step further than existing approaches (see Section 4), using the Software/Hardware Engineering (SHE) [8] real-time design methodology and the Parallel Object-Oriented Specification Language (POOSL) [8] [9] [10].

The paper is organized as follows. Section 2 gives some preliminary information necessary to understand this work, while Section 3 details the problem statement and Section 4 presents related research work. Section 5 shows the possibilities of expressing timing properties in a more convenient way and the scheduling analysis technique is presented in Section 6. A simple example is given in Section 7, while the last section presents our conclusions and ideas for future work.

2 Preliminaries

Software/Hardware Engineering (SHE) [8] is a system-level design methodology based on the formal modeling language POOSL (Parallel Object-Oriented Specification Language) [8] [9] [10], and on the code generation tool Rotalumis [11]. Using this methodology, models of real-time systems can be built and analyzed (Section 2.1). Moreover, the synthesis of an implementation out of a POOSL model is guaranteed to preserve the real-time properties, based on the ϵ-hypothesis (Section 2.2).

2.1 POOSL Modeling Language

POOSL is equipped with a mathematical semantics that can formally describe concurrency, distribution, communication, timing and functional features of a system in a single executable model, using a small set of very powerful primitives. Primitives can be combined in an unrestricted fashion and any combination has a precisely defined meaning. The formal semantics guarantees an unambiguous interpretation of a POOSL model, guided by semantical axioms and rules.

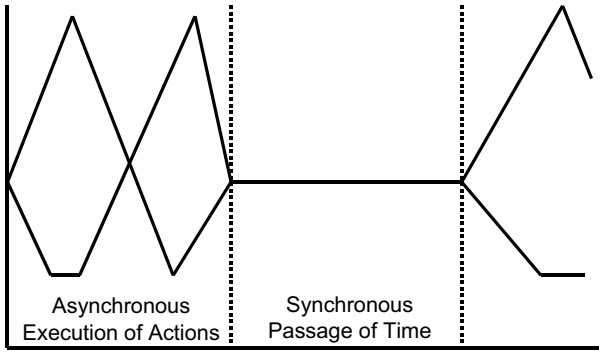

Fig. 1. Two phases of the execution of a POOSL model

POOSL consists of a process part and a data part. The process part (processes and clusters) is based on a real-time extension of the process algebra CCS [5] and is used for specification of real-time behavior of active components. The data part is based upon traditional concepts of sequential object-oriented programming and is used to specify the information that is generated, exchanged, interpreted or modified by the active components.

The language has a platform-independent semantics by defining a notion of *model time* without relying in any way on the clock of the underlying simulation platform. The semantics is based on a two-phase execution model (Figure 1): the state of a system changes either by asynchronously executing atomic actions,

such as communication or data computation, without time passing (phase 1), or by letting time pass synchronously without any action being performed (phase 2).

POOSL is able to capture both functional and timing properties of a system. Moreover, discrete-event approximations of continuous-time models of environment can be expressed, giving a unified way of modeling and reasoning about the properties of the system as a whole.

2.2 Properties Preservation by ϵ-Hypothesis

A real-time system can be formalized as a timed system consisting of a set of timed state sequences. A timed state sequence is a sequence of states with a time interval attached to each state. Each sequence represents a possible execution path of the real-time system. It is also called a timed execution trace.

The distance between two timed execution traces, if their lengths are equal (the same number of states/time intervals), is defined as the least upper bound of the absolute difference between the left-end points of the corresponding time intervals. Otherwise the distance between them is infinite. Introduced in [12], the ϵ-hypothesis guarantees that, if the distance between two timed execution traces is less than or equal to ϵ (Figure 2), then, if one of the traces satisfies a real-time property (expressed in $MITL_{\Re}$ formulas), that property, weakened upto ϵ, is preserved in the second trace as well.

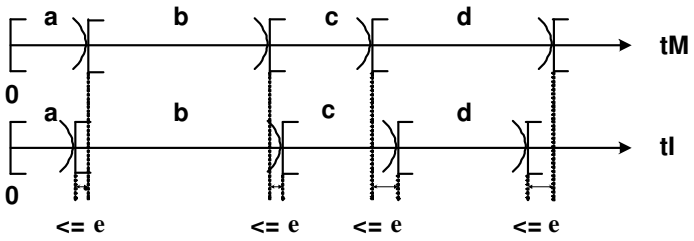

Fig. 2. Execution traces ϵ-close

In this paper, both model and implementation of a system are viewed as sets of timed execution traces. Actions in a model are timeless, while in reality, no matter how fast a processor is, it will always take a certain amount of time to execute them. Therefore, a *time-deviation* appears between model and implementation. If the code generation tool generates a set of execution traces ϵ-close to the traces in the model (thereby satisfying the ϵ-hypothesis), then all the properties of the model are preserved upto ϵ in the implementation [12].

In order to generate an implementation ϵ-close to the model, Rotalumis [11] synchronizes model time with physical time (Figure 3). All the actions that

happen in the model at a certain time t (`a` happens at model time `t1`, `b` at `t2`), are executed within a small ϵ amount of time around the corresponding moment t' in physical time (`a` is executed in $\epsilon 1$ around physical time `t'1`, `b` in $\epsilon 2$ around `t'2`). As a consequence, delays are not executed in the implementation exactly as specified in the model (`d` units of time): physical time passes until the next corresponding moment in model time is reached (the delay d is shortened to $d - \epsilon 1$).

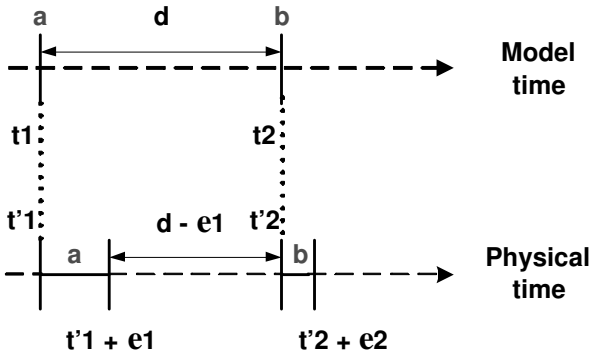

Fig. 3. Synchronization between model time and physical time

In [13] we have proposed a method for estimating the time-deviation between a model and its implementation on a single-processor target platform. The method is based on the Y-chart scheme (Figure 5) and on the possibility of POOSL to model and analyze, in a unified manner, application, target platform and their mapping as well. The time-deviation is determined by the maximum number of actions that is required to be executed at the same time in the model. If the value obtained for ϵ is considered too large, either the implementation must be generated for a higher performance platform, on which the execution of all the actions takes less time, or the model must be re-designed.

3 Problem Statement

In a POOSL model, functional behavior of a real-time system is specified by actions whose executions, according to the model semantics, are instantaneous (control actions). The semantics allows the specification of moments in time when each action should be executed, but it lacks the possibility of expressing the duration of an action, which makes a POOSL model unsuited for analyzing the scheduling of tasks.

The code generation tool, Rotalumis, guarantees the preservation of properties from a model to its implementation by satisfying the ϵ-hypothesis [12]. As

models specify that actions happen *now*, restrictions are imposed to the generation of code: the real-time properties can be preserved (upto ϵ) if the execution times of actions on the target platform are very small ($\ll \epsilon$).

These models allow proper reasoning about, and implementation of control-systems, in which actions typically take very small execution times. Nevertheless, it might yield a system over-specification: the real-time requirements could also be met if some actions required to happen at a certain moment, would be executed within some amount of time before a *deadline*. Imposing less restrictive timing constrains (actions can be executed *before* some moment, not necessarily *at* that moment) gives a lot more flexibility to the code generation tool. It also facilitates analysis and synthesis of systems with data computations, which typically take a considerable amount of time. If an action is modelled to happen at time `t2`, although at time `t1`<`t2` it can already start its execution, in the implementation the computation can start executing at `t1` and the result must be issued within ϵ around time `t2` (satisfying the ϵ-hypothesis). The real-time properties can be preserved if an appropriate scheduler facilitates these actions (computations) to meet their deadlines.

In this paper we propose a way of relaxing the timing constraints expressed in a POOSL model, which would allow the analysis of functional correctness *and* timing behavior of both control-systems and systems with time-intensive computations. Moreover, as far as the generation of code is concerned, the larger the number of actions that has to be executed at some moment, the more the preservation of properties is weakened. If some actions are allowed to start before their deadline, fewer are those that have to be executed at that moment, so the preservation of properties is weakened less severely. Our previous work [13], regarding the estimation of time-deviation for control-systems, can now be extended and applied for systems with computations as well.

4 Related Research

Classic scheduling theory [4], [14] enables analysis of timing behavior of a system and scheduling of its tasks onto the target platform so that the timing constraints can be satisfied. Real-time tasks (processes) are usually assumed to be periodic, i.e. tasks arrive (and will be computed) with fixed periodicity, and well-studied methods, like rate monotonic scheduling, are applied. Analysis of such models often yields pessimistic results and it is not able to handle non-periodic tasks with non-deterministic behaviors.

A way to relax the stringent constraints on task arrival times is proposed in [15], using automata with timing constraints to model task arrival patterns. The model is expressive enough to describe concurrency and synchronization of periodic, sporadic, preemptive or non-preemptive real-time tasks with or without precedence constraints. An automaton is schedulable if there exists a scheduling strategy such that all possible sequences of events accepted by the automaton are schedulable (all associated tasks can be computed within deadlines).

Based on the results obtained for schedulability analysis on timed automata, the TIMES tool [16] is designed for schedulability analysis and synthesis of real-time systems. A model consists of (i) a set of application tasks whose executions may be required to meet different timing, precedence and resource constraints, (ii) a network of timed automata describing the task arrival patterns and (iii) a preemptive or non-preemptive scheduling policy. From such a model, the TIMES tool can generate a scheduler and compute the worst-case response time for all tasks.

The timing constraint imposed on a task is the deadline relative to the task's release moment and it should be less than its worst-case execution time (WCET). In this sense, tasks modelled in the TIMES tool are equivalent to what we call computations in POOSL, while their timed automata are the equivalent of our timeless actions. But while the code generation tool for POOSL, Rotalumis, relies on the ϵ-hypothesis [12] for preserving the properties verified in the model, the TIMES tool relies on the synchronous hypothesis. In TIMES it is assumed that the time for handling system functions on the target platform can be ignored compared with the computing times and deadlines of tasks (the synchronous hypothesis).

5 Expressing Deadlines in POOSL

A typical timing constraint on a real-time task is the *deadline*, i.e. the time before which the task should finish its execution. In a hard real-time system, if the task's result is not ready before this moment, it is no longer useful. This may result in endangering people's safety or in a system that simply does not work properly.

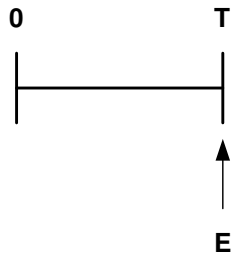

Fig. 4. The meaning of `delay T; E;`: action `E` takes no time

In a POOSL model, timing requirements of a real-time system are specified through `delay` statements that define the moments in time when actions should happen, while actions in the model do not take any time to be performed. In order to specify that an expression E must be evaluated at time `T`, one can write:

```
delay T; E;
```

According to the semantics of the language, the meaning of this model is: first delay `T` units of time and then instantaneously perform the computations required to evaluate expression `E` (Figure 4).

When code is generated out of this model, in order to preserve the properties of the system, `E` should be executed in ϵ time around moment `T` ($\epsilon \ll T$). This might not be possible if the time needed for the execution of `E` on a processor is comparable with `T`. In such a situation another interpretation of the model would allow the preservation of properties. If the meaning of the model was to start the evaluation of expression `E` at time `0` and to finish it before time `T` (`T` is the deadline for the execution of `E`), the generated implementation would preserve the properties of the model while still being consistent with it. But giving such an interpretation to different combination of statements in POOSL is not trivial.

Consider a simple example:

```
abort
    (delay T; E)
with p?urgentMessage;
```

The meaning of this model is that the execution of the `delay` followed by expression `E` can be aborted by the receiving of an `urgentMessage` on port `p`. The state of the model will reflect either that `E` has not started at all, or that `E` is already finished. In other words, at time `T` in the model, there is no possibility of being in an intermediate state, in which `E` has been only partly executed. If we interpret `T` as the deadline of `E`, then in the implementation the execution of `E` can start at time `0`. If an `urgentMessage` aborts `E`, the system ends up in a state which is not in the model and, from that moment on, the behavior of the system can be completely different from the behavior specified in the model. As a result, the interpretation given to the model, and consequently its implementation too, does not preserve the properties and deviate a lot from the semantics of the model.

This simple example already shows that the situations in which `delay`s can be interpreted as *deadlines*, are not so obvious. For the time being, we are still working to identify which exactly are the cases when a `delay` can be interpreted as a *deadline*, without violating the preservation of properties. Our final goal is to formalize them in such way that the code generation tool could automatically detect these situations and implement them accordingly.

6 Schedulability Analysis of POOSL Models

In a previous paper [13], we have shown how we can model the realization of a real-time system, using the Y-chart scheme concepts (Figure 5). A distinction is made between the software application (that describes the functional behavior of a system) and the platform (that eventually should execute the functional behavior), and the scheme allows exploration of different application-platform combinations.

An application model is usually described as a collection of communicating periodic tasks, and a platform model is specified as a collection of (parameterized) computation, communication and/or storage resources that are capable of executing the desired functional behavior [17]. Besides the model of the application, it is also required to model the physical devices (environmental model) that are controlled by the software and that exhibit a desired behavior [14]. Only the application model will be mapped on the platform model, but the environmental model is needed for reasoning about the system behavior as a whole.

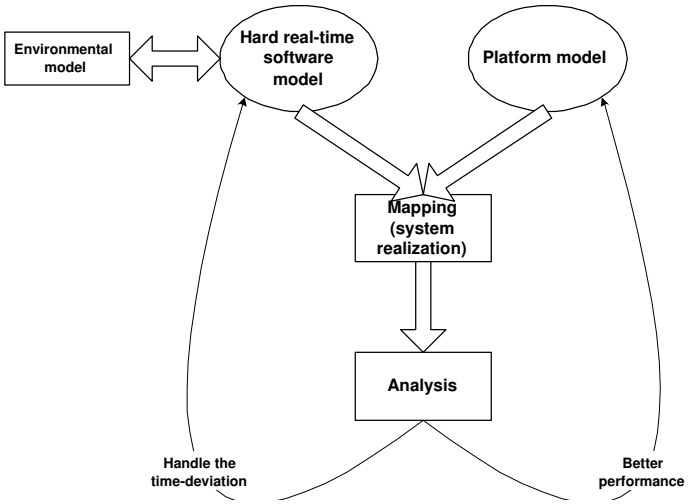

Fig. 5. Y-chart scheme for hard real-time systems

The mapping stage of the Y-chart scheme represents the assignment of each task to a computational resource that mimics the execution of its requests by time delays. Given the features of POOSL modeling language, the mapping consists in creating a communication channel between each task in the application model and the platform. Every time when an action is performed in a task, a message is sent through the communication channel to the platform model to inform what it has done. In this way the application model behaves as if the platform model was not present, while the platform model can estimate the time-deviation between the application model and its implementation. We call this mapping a model of the *system realization* because it gives a picture of how the system in reality would behave. The mapping *does not influence* the timing behavior of the application model: the platform is *informed* about the actions tasks *have done* and that in the real system *it* should perform, and the communication of a message does not take time.

If deadlines are specified for computations, it is possible that in the same time interval several computations happen in parallel, which means that they

must be scheduled on the underlying platform. Under these circumstances, the mapping stage of the Y-chart scheme is somewhat more complex than the one presented in [13]. Scheduling requests must be first sent to a scheduler to put them in an adequate ordering and to forward them to the target processor to be *executed*. The model of the processor receives these requests and estimates the time-deviation between the application model and its implementation on the real processor. Moreover, it informs the scheduler regarding each request whether it has managed to finish its execution or not. Based on this information, the scheduler analyzes the schedulability of the system.

To model deadlines for computations in the application model, a message is sent to the scheduler at the time a computation is allowed to start (at the moment there is no precedence constraint anymore), although in the model that computation will be performed later. The information contained in the message is the name of the computation (its WCET is known in the processor model) and its deadline:

```
schReq!execute(computation, deadline);
delay deadline;
computation()();
```

The scheduler is modelled as a POOSL cluster (Figure 6) that is always ready to receive requests from the application model components (modelled as POOSL processes). It arranges these requests in an adequate ordering according to a chosen scheduling policy. When a message is received from the application model, the scheduler knows that this particular computation can be started any time now. In order to keep the consistency between model and implementation, its execution must be finished before its deadline, which is the actual moment when the action happens in the model.

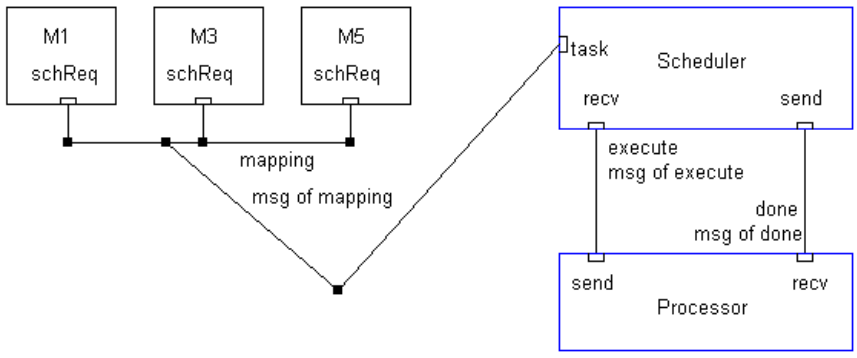

Fig. 6. Model of a real-time system

A processor (also modelled as a POOSL cluster) is described in terms of the worst-case execution times (WCET) of the computations required by the application model to be executed. It receives requests from the scheduler to *execute* different computations and it lets the time to pass according to their WCET. As the scheduling policy might be preemptive, the cluster modeling the processor is willing to stop at any time the *execution* of a particular computation in order to run another one:

```
abort
    (delay req getComputationTime())
with recv?execute(anotherReq);
```

The scheduler is informed about the status of the computations that were sent to execution. If a computation is not finished and its deadline has already passed, than the application is not schedulable, otherwise the computation is scheduled again.

Besides computations, there could also be some control actions: computations that take very little time and that are therefore modelled as instantaneous actions and are also meant to be interpreted in that way. If such an action appears in the system (arrives at the scheduler), it is considered urgent. It has priority of execution over the computations that have already been sent to the processor. At the processor level the execution of the current computation is interrupted and it is replaced by the control action whose execution is non-interruptible. The least upper bound of the amount of time needed for the execution of all the actions notified to the processor at time t is considered the time-deviation (ϵ) between model and implementation.

7 A Case Study

In order to emphasize the benefits of a unified model for the analysis of real-time systems, a simple case study has been modelled (Figure 6). The system is made of three periodic tasks (M1, M3, M5) that are modelled as follows:

```
task()()
    action(read)();
    schReq!execute(computation, deadline);
    delay deadline;
    computation()();
    action(write)().

periodic_task()()
    par
        task()()
    and
        delay period; periodic_task()()
    rap.
```

In an infinite loop, after reading the necessary information (which is modelled as a control action - `action(read)();`), computations may start immediately (`schReq!execute(computation, deadline);`). Their deadline is the same as the end of the period of the task, which is also the moment when the result is written (also modelled as a control action - `action(write)();`).

Table 1. Timing requirements in the case study.

Process	Period (ms)
M1	2.0
M3	3.0
M5	6.0

Table 1 presents the values of the periods and corresponding deadlines given in POOSL for the three tasks. The scheduling policy is earliest deadline first (EDF). At the processor level, the WCET for each of the computations and actions required by the application model is known (Table 2).

By simulating this application model, together with the scheduler and the model of the processor, besides the correctness of the system's behavior, we can analyze whether the system is schedulable or not, and moreover estimate the time-deviation ϵ between the application model and its implementation on the processor chosen. For the case study above we have obtained that the system is schedulable and the value of ϵ is 0.6 ns (there are at maximum 6 control actions that have to be executed at the same time).

Table 2. WCET known at the architecture level.

Process	Computation (ms)	Control Action (ns)
M1	0.5	0.1
M3	1.0	0.1
M5	2.5	0.1

8 Conclusions and Future Work

In order to achieve a predictable design of real-time embedded systems, a unified executable model, allowing analysis of both timing and functional aspects of a system, helps engineers in reasoning about different properties in a unified manner. Moreover, such a model should be easily refinable towards a complete system specification, from which the implementation can be automatically obtained.

In this paper we have presented an approach for tackling this issue, with the POOSL modeling language, the Rotalumis code generation tool, the SHE design methodology. A POOSL model can capture the functional behavior of a system in terms of instantaneous actions, for which it can also specify the moments in time when they happen. The timing constraints imposed in this way are very strict and the code generation tool is able to generate an implementation which preserves the properties from the model if its actions have very small execution times on the target platform. But for many systems small execution times cannot be obtained for all their actions. By introducing the possibility of expressing deadlines for computations, the timing constraints are more relaxed and the interpretation given to the model allows more flexibility for the generation of code. The time-deviation obtained between model and implementation is smaller, so the properties of the model are less weakened in the realization of the system.

As we have shown, it is not immediately obvious when `delay` statements can be interpreted as *deadlines*, without changing the behavior specified in the model. As future work, we aim at identifying all these situations and at formalizing them in such way that the code generation tool can automatically detect them and implement them accordingly.

References

1. Selic, B., Motus, L.: Using models in real-time software design. IEEE Control Systems Magazine **23** (2003) 31–42
2. OMG: Model Driven Architecture (MDA). OMG document ormsc/2001-07-01, Needham MA (2001)
3. Huang, J., Voeten, J.P., Ventevogel, A., van Bokhoven, L.J.: Platform-independent design for embedded real-time systems. In: Proceedings of the Forum on Specification & Design Languages 2003 (FDL'03). (2003)
4. Liu, C., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard real time environment. Journal of the Association for Computing Machinery **20** (1973) 46–61
5. Milner, R.: Communication and Concurrency. Prentice Hall International Series in Computer Science. Prentice Hall (1989)
6. Hoare, C.: Communicating Sequential Processes. Prentice Hall International Series in Computer Science. Prentice Hall (1985)
7. Alur, R., Dill, D.: A theory of timed automata. Theoretical Computer Science **126** (1994) 183–235
8. van der Putten, P.H., Voeten, J.P.: Specification of Reactive Hardware/Software Systems. PhD thesis, Eindhoven University of Technology, Eindhoven NL (1997)
9. Geilen, M.G.: Formal Techniques for Verification of Complex Real-Time Systems. PhD thesis, Eindhoven University of Technology, Eindhoven NL (2002)
10. van Bokhoven, L., Geilen, M., van der Putten, P., Theelen, B., Voeten, J., van Wijk, F.: POOSL formal modeling and specification language. ($http$: $//www.ics.ele.tue.nl/$ $lvbokhov/poosl/mainpoosl_n ocounter.shtml$)
11. van Bokhoven, L.J.: Constructive Tool Design for Formal Languages: From Semantics to Executing Models. PhD thesis, Eindhoven University of Technology, Eindhoven NL (2002)

12. Huang, J., Voeten, J.P., Geilen, M.C.: Real-time property preservation in approximations of timed systems. In: Proceedings of the First ACM & IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'2003). (2003)
13. Florescu, O., Voeten, J.P., Huang, J., Corporaal, H.: Error estimation in model-driven development for real-time software. In: Proceedings of the Forum on Specification & Design Languages 2004 (FDL'04). (2004) 228–239
14. Buttazzo, G.C.: Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. Kluwer Academic Publishers, Boston MA (1997)
15. Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: TIMES - a tool for modelling and implementation of embedded systems. In: Proceedings of 8th International Conference, TACAS 2002, Springer-Verlag (2002) 460–464
16. Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: TIMES: a tool for schedulability analysis and code generation of real-time systems. In: Proceedings of the 1st International Workshop on Formal Modeling and Analysis of Timed Systems, FORMATS 2003. (2003)
17. van Wijk, F.N., Voeten, J.P., ten Berg, A.: An abstract modeling approach towards system-level design-space exploration. In: System Specification & Design Languages (Best of FDL'02). Kluwer Academic Publishers, Dordrecht NL (2003) 267–282