Modeling and Simulating Functional and Timing Aspects of Real-Time Systems by Delegated Execution

Martin Naedele*
Computer Engineering and Networks Laboratory (TIK)
Swiss Federal Institute of Technology Zurich
CH-8092 Zürich, Switzerland
naedele@tik.ee.ethz.ch

Abstract

Scheduling simulators are useful in the design exploration phase of the development of reactive embedded real-time computer systems. However, most conventional scheduling simulators do not fulfil all the requirements needed for the modeling of application functionality and system environment. Other system modeling approaches exist for this purpose, but these usually do not deal with scheduling aspects.

This paper proposes an new approach to the modeling and simulation of both functional and scheduling aspects of real-time systems. The new approach is based on Petri nets and uses a principle called delegated execution. *The main idea is to associate time progress with an explicit scheduler subnet instead of the task structure. Advantages of this approach are the ability to model preemptive scheduling, to combine an application with several schedulers, and to reuse standard scheduler models.*

The modeling methodology is demonstrated using the well-known mine drainage system case study.

1 Introduction

Real-time systems are computer based systems for which both functional correctness and timing correctness need to be guaranteed in order to achieve the intended purpose. The successful design of an embedded real-time system requires an unambiguous system specification. Even though one would in general prefer to be able to formally analyze the correctness of a real-time system specification, there are certain situations, especially with respect to reactive systems, in which an executable specification, which allows simulation, is nevertheless useful [20, 1, 9, 3]: Analytical results, e.g. schedulability tests, give only binary answers but do not guide the designer towards a better solution. An executable specification can be used to evaluate the performance of the specified system. Simulations are frequently used to evaluate the performance of new algorithms, especially for systems with aperiodic tasks. Certain systems may be too complex (e.g. overhead, overload, interaction with an environment for which no analytical models exist, conditional dependencies, distributed systems, imprecise computation, self-modification of the system, fault simulation) for the analytical means available today - even though there are some promissing approaches recently [11] - so that simulation of the specification is the only way to gain at least some confidence in the correct functioning.

A scheduling simulator for embedded reactive systems should allow the specification of the

1. application (task structure, task functionality as far as needed to allow for conditional dependencies, timing parameters, and system environment),
2. execution architecture (single or multiprocessor, communication protocols for distributed systems), and
3. scheduling algorithms.

In order to be useful for the exploration phase of system design when many changes will be made and experts of different domains are working together to deliberate on system tradeoffs, it is also necessary to have

4. a intuitive, domain independent and high-level specification formalism, preferably a visual modeling language,
5. concepts for hierarchical structuring, abstraction and refinement,
6. facilities for animated execution for model debugging,

*The author is now with ABB Corporate Research Ltd, CH-5405 Dättwil, Switzerland.

7. a simple yet formal execution semantics for the formalism used.

Conventional scheduling simulators (e.g. [1, 20, 9, 15]) do not fulfil all the requirements stated above: STRESS [1] is a textual simulation language with language features and evaluation tools geared towards evaluation of real-time systems. PERTS/DRTSS [20, 21] is a C^{++} framework for the simulation of real-time systems. Its main feature is the so-called live task model which allows to hook actual implementations of tasks into the simulator structure. DRTSS has a graphical frontend to define simple task structures, but most structural attributes even there need to be entered as textual descriptions. ASSERTS [9] comes closest to the separation of structural/functional and timing attributes used in this paper, though also only with a textual specification of the task structure. It is not clear in how far the scheduler and architecture specifications can be customized. None of these scheduling simulators has formally defined execution semantics, none offers full visual model construction and animated execution, and there are only limited facilities to model the system environment.

On the other hand, several different approaches exist for the visual specification of real-time systems: State machine based approaches (StateCharts [10, 22], ROOM [19]) can not express data flow and have no agreed upon execution semantics. Informal specification languages like SA/RT or HRT-HOOD [6] are not executable and behaviorial aspects are usually expressed as state machines. Petri nets and their derivatives are the only formal specification language to combine the ability to express control flow and data processing, concurrency, synchronization, and intuitve formal execution semantics, which allows formal analysis under certain conditions, with a clear and simple visual representation.

However, Petri net based approaches concentrate on the functional correctness of real-time systems and generally neglect low-level timing as well as shared resources and lack means to incorporate the scheduling strategy in the model in a general way. [3] introduces an object-oriented temporal high-level Petri net formalism and demonstrates how a real-time system and its environment can be modeled with it. The approach targets system simulation and automatic application generation. [7] has a similar approach, but without any notion of time at all. In [5] a temporal Petri net formalism is presented that is both expressive and analyzable, but can not model preemption. None of these works deals with the issue of scheduling, or even preemptive scheduling, and resource access protocols. In [17] Petri nets are used to represent state-machines. Though it is suggested to make the scheduler part of the model, no general rules for the construction of schedulers are given. The demonstrated approach can not express preemption and the scheduler is an integral part of the model with no clear interface and thus can not be exchanged easily.

To incorporate scheduling is difficult because for system exploration one must find a way to associate different scheduling strategies with the model of the real-time application. One solution is to modify the underlying Petri net firing semantics [2]. However, this is not transparent to the modeler, especially when experimenting with different schedulers. In addition, it loses the advantage of being able to use the formal theory of Petri nets for analysis purposes, and does not solve the problem of expressing preemption. A better solution is to make the scheduler an explicit part of the system model. However, previous attempts to do this run into the problem of problem of n^2 connection complexity when trying to model preemptive schedulers [12, 17].

This paper presents a new integrated approach to functional and timing simulation which fulfils all the requirements stated above. In particular, application, architecture, schedulers, and even the simulation analysis functionality can be described with the same visual modeling paradigm, while the presented modeling approach provides separation of concerns with a clear interface between application and scheduler submodels. The approach can be used with any of the tools for high-level Petri nets as described in section 3 which avoids the dependence on a particular tool. The feasibility of the suggested approach is demonstrated with a case study of the well-known mine drainage system [6, 17]. This case study has been conducted with the CodeSign system modeling tool which offers visual editing, animation, and instrumentation/analysis functions, based on an object-oriented time high level Petri net formalism [8] with hierarchical structuring.

As additional result, it is shown for the first time how general preemption can be modeled with high-level time Petri nets without further formalism extensions. Based on this result, a library of interoperable Petri net models of scheduling algorithms has been developed.

2 Delegated execution

Up to now, approaches that use Petri nets to specify real-time systems largely ignore timing aspects, such as scheduling. Presumably, the main reason for this is that previous approaches, as far as they are concerend with temporal aspects at all, associate the consumption of execution time of a block with the transition that represents this block. This approach, called *task-centered execution*, is intuitive but is impractical for modeling fully preemptive scheduling, as all tasks would need to be divided into atomic blocks with connections between the atomic blocks of all tasks.

This paper suggests a novel approach: *delegated execution*. The main idea of this approach is that the model consists of two parts: On the one hand, the application task structure which describes the functionality and the interaction but has only untimed transitions that represent the start and completion of task or task segment. On the other hand, the scheduler/processor which determines the actual delay between release and completion for all task segments. Thus the determination of the actual completion time is delegated to the scheduler. The scheduler is the only part that needs to have knowledge about the partial completion of each task, thus allowing scalable modeling of preemption.

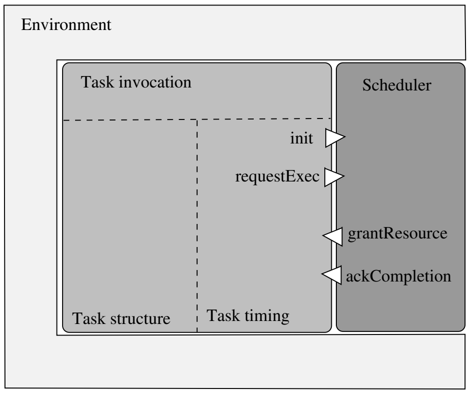

Figure 1. Basic blocks of the proposed generic model structure.

Fig. 1 shows the generic structure of the resulting system model. The application specific part of the model (shown on the left) consists of the environment model, above mentioned *task structure* with intertask communication, resource access, and data processing, and additionally the *task invocation* and *task timing* submodels which are responsible for controling task releases and the conversion of task releases into parametrized (e. g. deadlines, priorities) requests for the scheduler.

The scheduling and resource allocation policies are standard algorithms and thus a library of subnets for the scheduler subsystem can be developed for use with any system model (see Section 5).

In order to be able to use such predefined scheduler subnets, the following interface is defined between the task timing subnet and the scheduler subnet: `init` for task set dependent initialisations of the scheduler (e. g. priority ceilings), `requestExec` for execution requests, `grantResource` for notifications that the scheduler has allocated a certain resource, and `ackCompletion` for the acknowledgement that a certain task segment has been completely executed. The resource grant and completion acknowledgement tokens from the scheduler contain task id values, so that a structural conflict between the task transitions waiting for these tokens can be resolved deterministically.

As this is an event based interface with a clear definition of the information to be passed over the interface [14], it is even possible to connect it to scheduler modules not implemented in Petri nets but other formalisms or programming languages.

In the suggested modeling methodology, first the task structure subsystem and the task invocation subsystem (environment) are developed and analyzed to assure correct functional behavior of the system. Then the task timing subsystem is added to the task body structure and all timing related design parameters are specified there. Now different schedulers are connected to this interface. Through scheduling simulation, the timing behavior of the system is evaluated, and, if necessary, a different scheduler is used or timing parameters are adjusted.

3 Petri net model of a preemptive scheduler

Introductory information about Petri nets, which has to be omitted here due to lack of space, and a more detailed discussion of various formalism extensions can e.g. be found in [13]. The example of a preemptive fixed-priority scheduler in this section will be used to explain both the Petri net formalism used in the case study and how to model preemption with Petri nets.

Fig. 2 shows the model of a preemptive fixed-priority scheduler with neither resource allocation, nor any provisions for non-ideal behavior, such as context switching or scheduler computation overhead. Detailed descriptions of models of more complex scheduling algorithms (see Section 5) can be found in [14]. Of course, the approach is not restricted to single-processor models: Like the application submodel also the scheduler submodel may be hierarchically structured, e.g. to represent distributed architectures.

The delegated execution approach requires the use of high-level time Petri nets [8] (e.g. Colored Petri Nets) for the scheduler. High-level Petri nets have not only 'black' tokens, but their tokens can carry data structures. In Fig. 2 the data type (class) of the tokens on a place is stated within the brackets. The task attribute data structure, for example, has the fields priority (pri), remaining execution time (c), and start time of the current slice (ts). The enabling of a transition can be *guarded* by a condition on the values of the enabling tokens. The *action* part of a transition allows to process token data values. For the inscription language used in this paper, the code of both guards and actions is executed sequentially, the expressions address tokens by the variable names on the arcs, and the 'dot notation' common for object-oriented languages is used to access the fields and methods of tokens.

With time Petri nets [4], a firing *delay* may be specified for a transition. The transition then has to be enabled continuously for the duration of this specified delay before it fires. In contrast to the usually used timed Petri nets [18], with time Petri nets the waiting can be aborted by removing

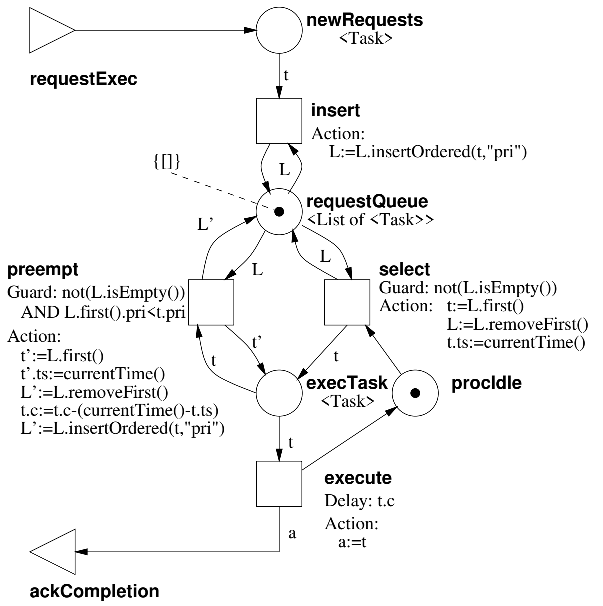

Figure 2. Basic preemptive fixed-priority scheduler.

one of the enabling tokens from its place.

In order to structure the system model and to be able to easily exchange submodels, such as the scheduler, a Petri net formalism with a concept of composition and hierarchy is useful. In this paper the component concept of [8] will be used: Components are subnets augmented with input and output ports (graphically represented as triangles). Input ports write into places, output ports are written into by transitions. Ports indicate the places where arcs entering or leaving certain subnets are to be fused. As described in [8], the component concept can be combined with object-oriented concepts to faciliate component reuse and the construction of parameterizable models.

The basic scheduler of Fig. 2 uses only two of the interface ports described in section 2. A priority ordered queue of runnable tasks is stored in the list token in place `requestQueue`. It is assumed that the list token class has a method (`insertOrdered()`) to insert elements into a list, sorted with respect to a specified key field, e. g. the priority.

As a time Petri net is used, preemption can be realized by removing the token from the place `execTask` and thus disabling the firing of the delayed transition `execute`. In order to achieve proper task progress in the presence of preemption the remaining execution time is calculated, for which access to the current simulation time is necessary, and stored in the token representing the preempted task. This happens in the transition `preempt`. By storing the timer state in the token and using the token value to specify the transition delay, instead of storing the remaining delay in the transition itself [12], it is possible to express preemption of arbitrary tasks and to do so without any need for replication of the 'CPU transition' `execute`.

The guard condition and action code for the `preempt` transition may easily be extended to give priority to aperiodic tasks, or to discriminate between preemptable, non-preemptable, and non-resumable behavior of different tasks. Similarly, it is straightforward to add a delay to the `preempt` and `select` transitions to incorporate context switching overhead and to take a processor speed factor into account when calculating the delay of `execute` from the remaining execution time of the task in `execTask`.

In this scheduler example preemption is triggered by the arrival of a higher priority task. In a similar way it would be possible to construct a scheduler model where the executing task is preempted in regular intervals to model e.g. a time-slice scheduler.

4 Case study: mine drainage system

4.1 Specification and overview

In this section the well-known mine drainage system (e.g. [6]) is modeled with Petri nets using the approach described above. Parts of the resulting model will be used as examples for how to construct the application specific part of a system model. A more detailed and systematic explanation of how to model different situations can be found in [14]. The purpose of the drainage system is to pump out the water that collects at the sole of the mine and to monitor several environmental parameters in the mine, such as water, CO2, and CH4 levels, as well as air and water flow. As soon as the water level reaches a certain mark, the pump is activated if the CH4 level is not too high. Detailed specifications for the mine example application can be found in [6].

Fig. 3 shows an overview of the application task structure of the mine drainage system with non-transparent components containing subnets. Not shown is only the scheduler (see Section 3) which would be connected to the interfaces on the right hand side. In the chosen implementation, the application has seven tasks (airflow monitor, CO2 monitor, waterflow monitor, CH4 monitor, interrupt handler, pump motor on, and pump motor off). CH4 monitor and 'motor on' communicate via a shared memory location for the `CH4StatusOK` flag, and both motor tasks have to coordinate their access to the motor itself (Place `pumpRunning`).

The elements in the upper right hand corner of the application subnet are the sump, that is the environment that

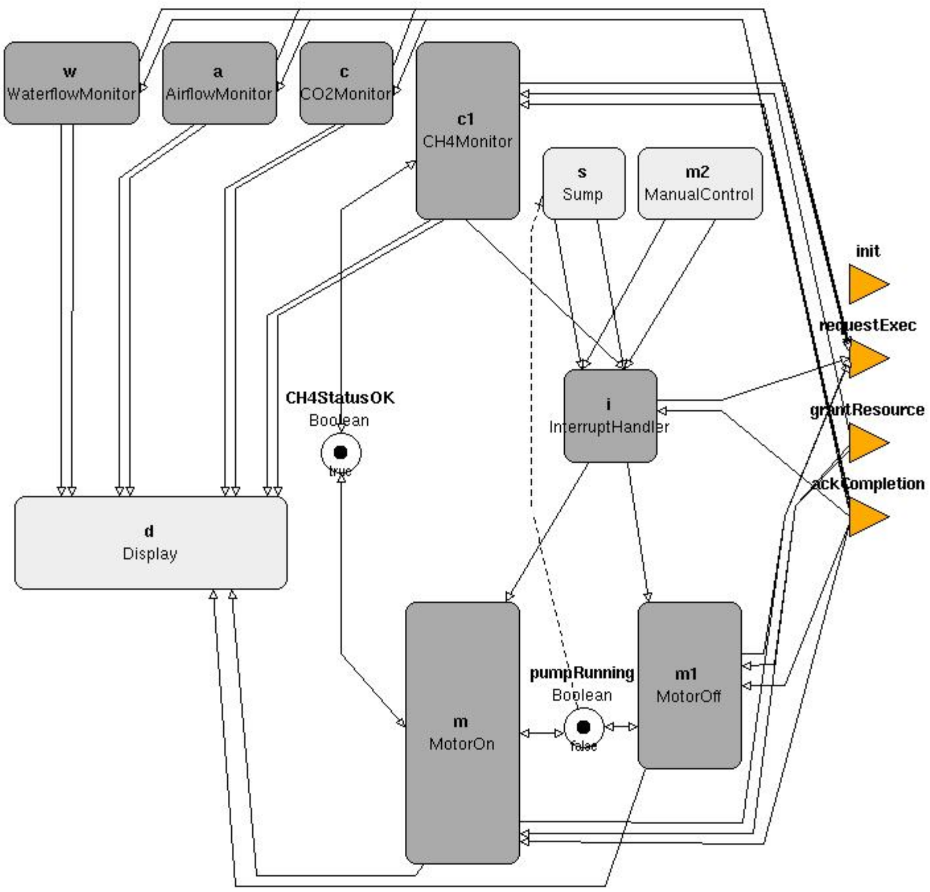

Figure 3. The application task structure sub-model for the mine drainage system example (screenshot of the model in the CodeSign tool). In this and the following figures a dashed line indicates a read arc. Note that the interface ports all point in the same direction because the CodeSign tool can not visually distinguish inputs and outputs.

Task	pri	C_{wc}	per	d_{rel}	resources
Interrupt handler	2	0.11	100	*	
CH4 monitor seg. 1	3	0.15	5	*	
CH4 monitor seg. 2	3	0.01	5	*	
CH4 monitor seg. 3	3	0.09	5	*	ch4StatusOK
Motor off seg. 1	4	0.1	-	*	
Motor off seg. 2	4	0.06	-	*	motor
CO2 monitor	6	0.15	60	1	
Airflow monitor	7	0.15	60	2	
Waterflow monitor	8	0.15	60	3	
Motor on seg. 1	9	0.1	-	*	ch4StatusOK
Motor on seg. 2	9	0.16	-	*	motor

Tab. 4.1: Mine Drainage System Task Parameters. The interrupt handler is an aperiodic task, so the value given as period is the minimum inter-arrival time. For the deadline entries with a '*' refer to the text.

generates high and low water level interrupts, and the operator switches (on/off) for the pump. The display component in the left lower corner represent the alarm indicator and data logging facilities.

Tab. 4.1 lists the task parameters, which are in most cases the same as in [6]. The specification dictates two end-to-end deadlines: First, the sequence interrupt - 'interrupt handler' - 'motor on' must execute within 20 seconds, and second, the sequence 'CH4 monitor' - 'interrupt handler' - 'motor off' must execute within 1 second.

4.2 Task structure

Fig. 4 shows the subnet within the 'CO2 monitor' component. The Petri net shown is the result after all modeling steps (see Section 2) have been completed. It is assumed

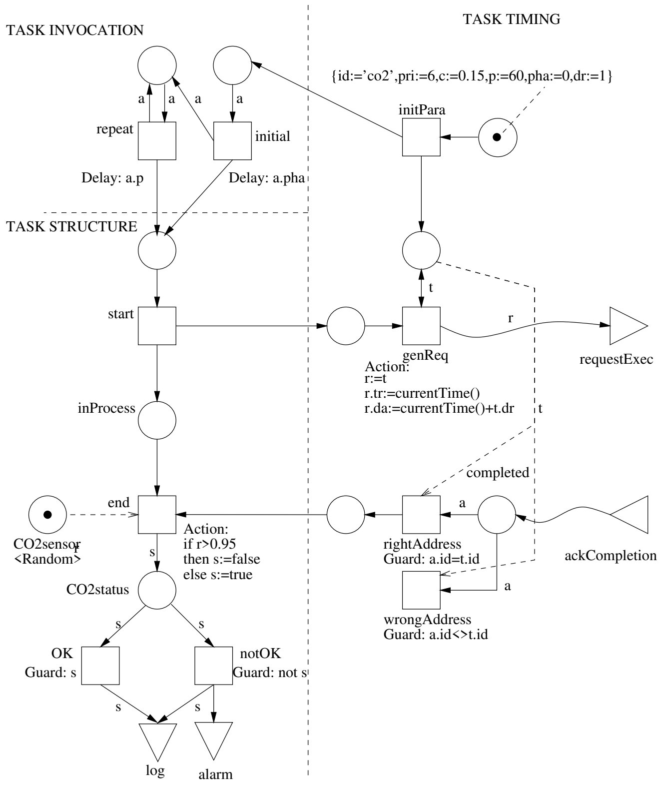

Figure 4. The task model for the CO2 monitor task.

that standard Petri net analysis methods have been used to verify functional correctness before the task timing information was added. The `initPara` transition initializes the task invocation subnet and the task timing subnet with the task attributes like phase, period, task id, execution time, and deadline. The task invocation subnet for a periodic task consists of one transition for the initial phase delay and another for the periodic reoccurence of the request.

The CO2 task consists of only one task segment which starts with transition `start` and ends with transition `end`. `start` sends a black token to the `genReq` transition of the task timing subnet. `genReq` generates the request token from the task attributes. Here, the task id is 'co2', the priority is 6, the actual execution time is 0.15 units and the relative deadline is 1 unit. The transition sets the release time and the relative deadline attributes, and passes the request token on to the scheduler via the `requestExec` interface. The transition `rightAddress` fires when it recognizes among the task completion acknowledgements from the scheduler one that indicates the completion of the CO2 task. As a result, the `end` transition in the task structure subnet may fire. The remaining net elements emulate the reading of a sensor value and activate the logging and/or alarm facilities depending on the sensor value read.

The delegated execution model assumes that each execution request concerns one block of task execution for which it is not important at what time exactly between request and completion the computation is executed. In the case that the task body contains actual computation instructions as transition code, this code must not be written into the transition that sends the execution request (`start`), because then the calculation would be made before the task segment was scheduled and processed. However, all results will be correct if the action code belonging to a segment is executed at the same time as the transition that synchronizes on the completion acknowledgement signal. If necessary, the application code has to be distributed into several segments.

The airflow and waterflow monitor tasks are identical, except for different task attributes. Interrupt handler is an aperiodic task. Its task invocation submodel is simply an input for a token from one of its three interrupt sources.

The three remaining tasks access shared resources and are divided into several segments:

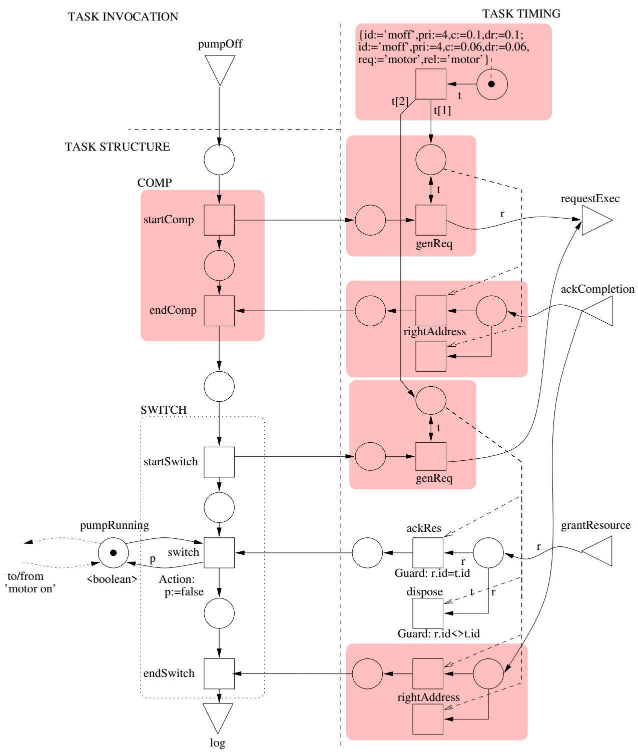

Figure 5. The task model for the 'motor off' task with shared resource.

The 'motor off' task (Fig. 5) is an example for an aperiodic task with two segments (*comp* and *switch*), the second of which accesses a shared resource, in this case the I/O port to control the motor which is shared with the 'motor on' task. The shaded parts of the figure indicate patterns that are identical to the ones presented in Fig. 4. Their inscriptions have been omitted for clarity. The second segment contains another place and transition `switch` to account for the fact that the actual access to the resource in the application model (place `pumpRunning`) must happen only after the resource has been granted to the task by the scheduler but before the segment is completed and the resource is released again.

The 'CH4 monitor' is a periodic task with three segments and one shared resource and 'motor on' is an aperiodic task with two shared resources (motor control and CH4 status) and two real segments.

Clearly, using Petri nets to describe task data flow and control structure, it is straightforward to express not only simply precedence chains but also nonlinear control flow (AND, OR), loops, imprecise computation, recovery blocks etc.

Compared to [17], the new approach results in a model with less stuctural overhead, better congruence of application and model structure, and better modularization and separation of concerns, which makes it easier to understand, maintain and modify. Furthermore, the approach can express fully preemptive scheduling and shared resources. As could be seen from the examples, especially Fig. 5, the task structure and in particular the task timing subnet consist of recurring structures (patterns [16]). A suitable net editing tool may automatically generate the corresponding templates from the timing parameters and also offer options to hide the structural parts (task timing subnet and connections to the scheduler) in order to facilitate the understanding of the application specification (task structure and task invocation subnets).

4.3 Simulation results

Figures 6 and 7 show the resulting Gantt charts[1] for the critical scheduling situations for simulations with preemptive fixed-priority scheduling with priority ceiling and non-preemptive FCFS scheduling on a single processor.

The most critical situation arises whenever the 'CH4 monitor' executes and initiates an emergency shutdown of the motor while a 'motor on' sequence is also executing. For the fixed-priority PCP scheduler this worst case occurs when the 'CH4 monitor' task is activated during the execution of the first segment of 'motor on'. With non-preemptive FCFS scheduling, an even worse case with respect to the shutdown deadline can be constructed when the interrupt handler for the 'motor on' sequence is released slightly before the three simple monitor tasks, and the the 'CH4 monitor' just after the interrupt handler has requested the 'motor on' task.

As can be seen from the figures (the time scale in the middle of the chart has a minor spacing of 0.1 seconds), the end-to-end deadline for the shutdown sequence is met for the fixed-priority with PCP scheduler, because the maximum blocking time the shutdown sequence can encounter is the length of 'motor on'. However, for non-preemptive FCFS scheduling the time span from the release of the CH4 monitor ($0.11+\epsilon$ s) to the end of 'motor off' (1.34 s) is 1.23 seconds and the deadline (1.11s) is missed. This example shows that the simplified assumption of non-preemptive and random scheduling often made in the Petri net modeling of real-time systems [17] may lead to wrong results. Results for other schedulers (EDF, fixed priority) can be found in [14].

Even though the mine drainage system is only a relatively small and simple example (about 120 places, 105 transitions, and 380 edges), it exhibits features that, taken all together, prohibit the use of formal verification methods: non-deterministic environment (plant), data processing and data dependent branching decisions, preemptive scheduling, and access protocols for shared resources.

5 Conclusion

A new approach to the modeling of real-time systems with Petri nets, *delegated execution*, has been presented. In contrast to previous work, this approach allows to model and simulate both the functional and the timing behavior of a system with fully preemptive scheduling and shared resources. A generic system model has been developed that clearly separates application function, timing parameters, and scheduling algorithm.

To demonstrate the applicability of the proposed approach, the following scheduling algorithms were implemented and used for scheduling simulations with the Petri net modeling and simulation tools developed in the Moses project [14][2]: non-preemptive first-come-first-served (FCFS), preemptive fixed-priority, earliest-deadline-first (EDF), least-laxity-first (LLF), non-preemptive shortest-job-first (SJF), timer-driven fixed priority, integrated interrupt driven fixed priority, non-integrated interrupt driven fixed priority, polling server, deferrable server, priority exchange server, basic priority inheritance protocol, priority ceiling protocol, and stack resource policy.

Currently we are enhancing our tool set to increase simulation speed and model construction comfort. In future work, we will collect and document a set of Petri net design patterns [16] that would facilitate the construction of

[1] Produced with the Moses tool set, see section 5.

[2] The model library and the Petri net tools are available from `http://www.tik.ee.ethz.ch/~moses/download.html`

real-time system Petri net models. We are also looking into ways to integrate Petri net based executable system specifications with other software modeling formalisms, such as the Unified Modeling Language (UML).

References

[1] N. Audsley, A. Burns, M. Richardson, and A. Wellings. Stress: a simulator for hard real-time systems. *Software - Practice and Experience*, 24(6):543–564, June 1994.

[2] S. Balaji, L. Patnaik, L. Jenkins, and P. Goel. S-nets: A petri net based model for performance evaluation of real-time scheduling algorithms. *Journal of Parallel and Distributed Computing*, 15:225–237, 1992.

[3] M. Baldassari and G. Bruno. A methodology and environment for the object oriented analysis and design of real time systems. In *Proceedings Euromicro Workshop on Real-Time Systems*, 1990.

[4] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, Mar. 1991.

[5] G. Bruno, A. Castella, I. Pavesio, and M. Pescarmona. A new Petri net based formalism for specification, design and analysis of real-time systems. In *Proceedings IEEE Real-Time Systems Symposium*, 1993.

[6] A. Burns and A. J. Wellings. HRT-HOOD: A structured design method for hard real-time systems. *Real-Time Systems Journal*, (6):73–114, 1994.

[7] C. Ebert. Experiences with colored predicate-transition nets for specifying and prototyping embedded systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part B,*, 28(5):641–652, Oct. 1998.

[8] R. Esser. *An Object Oriented Petri Net Approach to Embedded System Design*. PhD thesis, ETH Zurich, 1996.

[9] K. Ghose, S. Aggarwal, P. Vasek, S. Chandra, A. Raghav, A. Ghosh, and D. Vogel. Asserts: A toolkit for real-time software design, development, and evaluation. In *Proceedings Euromicro Workshop on Real-Time Systems*, 1997.

[10] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[11] H. Kwak, I. Lee, A. Philippou, J. Choi, and O. Sokolsky. Symbolic schedulability analysis of real-time systems. In *Proceedings IEEE Real-Time Systems Symposium*, 1998.

[12] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modeling with Generalized Stochastic Petri Nets*. John Wiley & Sons, 1995.

[13] T. Murata. Petri nets: Properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr. 1989.

[14] M. Naedele. Petri net models for single-processor real-time scheduling. TIK-Report 61, ETH Zurich, Computer Engineering and Networks Laboratory (TIK), 1998.

[15] M. Naedele. A survey of real-time scheduling tools. TIK-Report 72, ETH Zurich, Computer Engineering and Networks Lab (TIK), Apr. 1999.

[16] M. Naedele and J. W. Janneck. Design patterns in Petri net system modeling. In *Proceedings ICECCS'98*, 1998.

[17] L. Nigro and F. Pupo. Modeling and analysing DART systems through high-level Petri nets. In *Application and theory of Petri Nets 1996*, number 1091 in LNCS, 1996.

[18] C. Ramchandani. Analysis of asynchronous concurrent systems by timed Petri nets. Technical Report 120, Project MAC, Massachussetts Institute of Technology, Feb. 1974.

[19] B. Selic, G. Gullekson, and P. Ward. *Real-time object-oriented modeling*. John Wiley and Sons, 1994.

[20] M. Storch and J. W.-S. Liu. A simulation environment for distributed real-time systems. Technical report, Department of Computer Science, University of Illinois, Urbana, 1994.

[21] Tri-Pacific Software, Alameda. *PERTS/PERTS Sim User Manuals*.

[22] M. von der Beeck. A comparison of Statecharts variants. In *Proceedings third international symposium on formal techniques in real-time and fault-tolerant systems*, number 863 in LNCS, 1994.

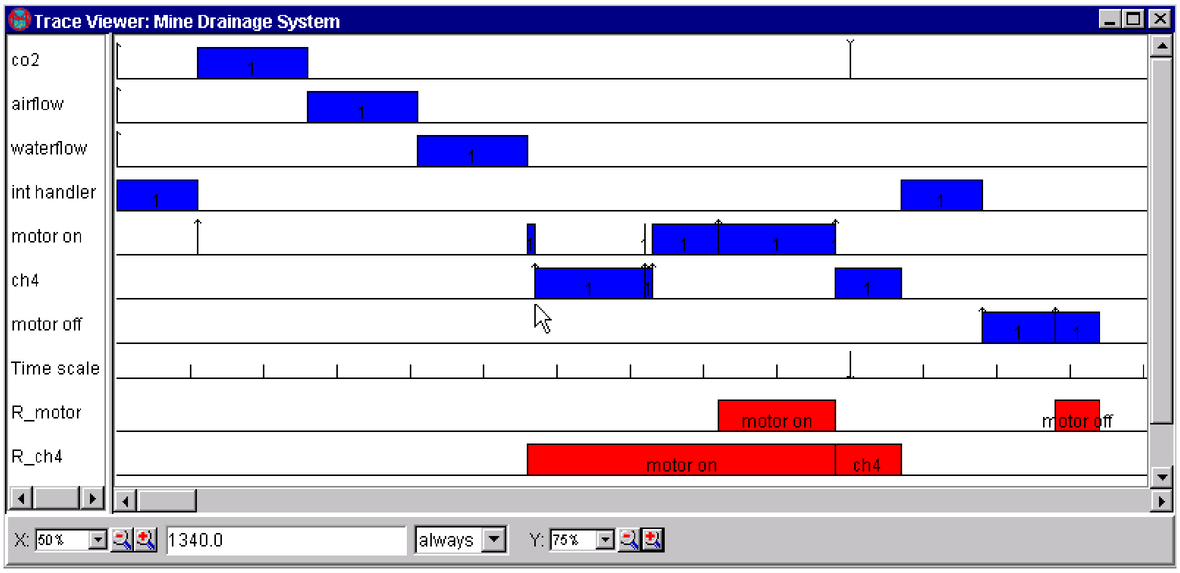

Figure 6. Simulation with preemptive fixed-priority scheduling and priority ceiling protocol (first 1.4 s). The arrow indicates the release of the CH4 monitor task. Above the time scale (divisions: 0.1 s) the task requests, deadlines, and execution intervals are drawn, below it the locking intervals and locker tasks of the resources are visualized.

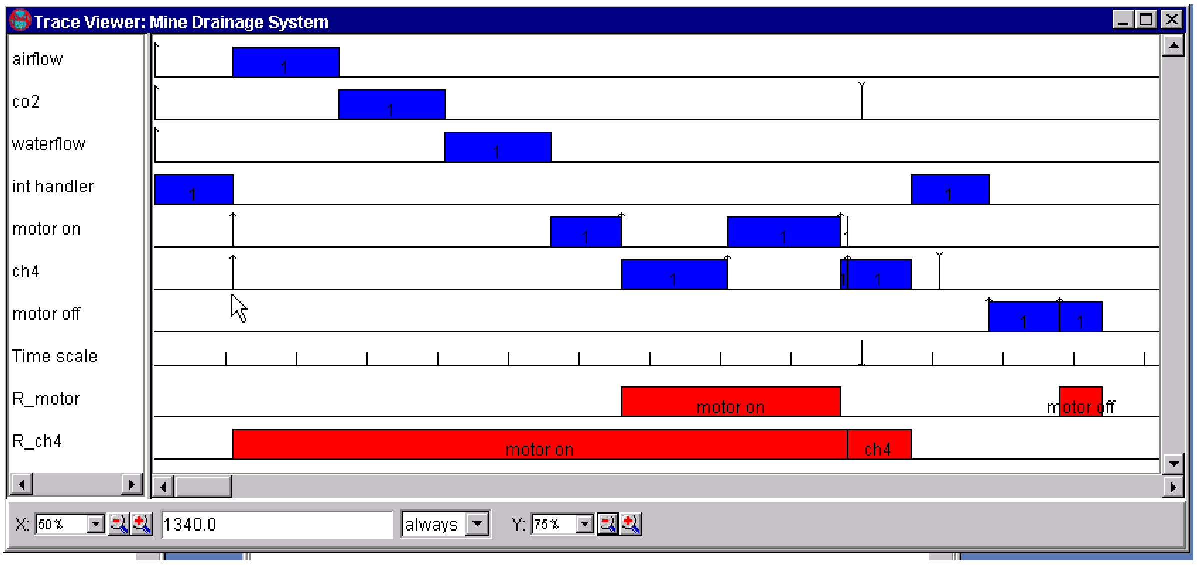

Figure 7. Simulation with non-preemptive FCFS scheduling (first 1.4 s).