Proceedings of the 17th World Congress
The International Federation of Automatic Control
Seoul, Korea, July 6-11, 2008

System-level Verification of Multi-Core Embedded Systems using Timed-Automata ⋆

Jan Madsen * **Michael R. Hansen** * **Kristian S. Knudsen** *
Jens E. Nielsen * **Aske W. Brekling** *

* *Informatics and Mathematical Modelling, Technical University of Denmark, Richard Petersens Plads, DK-2800 Kgs. Lyngby (e-mail: {jan,mrh,awb}@imm.dtu.dk).*

Abstract: A key challenge of implementing an embedded systems application on a heterogeneous multi-core platform is to find the right mapping of the application onto the execution platform. The right mapping is dependent on the characteristics of the platform, i.e. processors and the network connecting them, as well as the application. As embedded systems are heavily resource constrained and often safety-critical, there is a strong desire to be able to reason about properties of the system at an early stage in the design process, i.e. at the system-level. In this paper, we present a system-level modelling framework which allows for cross-layer modelling and verification, covering the application layer, middelware layer (RTOS), and hardware layer. The modelling framework allows the designer to verify the impact of execution platform and application mapping on the schedulability (meeting hard real-time requirements), power consumption and memory utilization, while taking communication into account. The modelling framework is implemented using timed-automata in UPPAAL, Behrmann et al. [2004] and the feasibility of the framework is illustrated through a case-study of a real-time multimedia application consisting of 3 applications with a total of 103 tasks executing on a platform with 4 cores.

1. INTRODUCTION

Embedded computer systems are making their way into more and more devices, from household appliances to mobile phones, PDAs and cars. Many of these systems have a limited amount of resources such as memory and power, and must perform in a timely manner imposed by their application domain.

As it becomes harder to improve computer performance using sequential execution, the trend moves toward using multi-core system designs, integrating multiple processing units connected through a network. One or more applications are divided among these processing elements. As these systems become more complex, the interaction between application and execution platform, becomes more incomprehensible and problems such as memory overflow, data loss and missed deadlines become more likely. In the development phase it is not enough to simply look at the different layers of the system independently, as a minor change at one layer can greatly influence the functionality of other layers. System-level verification of schedulability, upper limits for memory usage and power consumption, taking all layers into account, has therefore become a central field of study, in designing real-time systems.

As many important design decisions are made early in the design phase, it is imperative to support the system designer at this level. This paper presents an abstract embedded system model which is able to capture a set of applications executing on a multi-core execution platform. This model has been formalized with timed-automata semantics using UPPAAL, Behrmann et al. [2004]. The formal semantics gives the ability to model-check properties of timing, memory usage and power consumption.

In order to support designers of industrial applications, the timed-automata model is hidden for the user, allowing the designer to work directly with the system-level model. The designer provides an application consisting of a task graph (possible a set of), an execution platform consisting of processing elements interconnected by busses and a mapping of tasks to processing elements. The system model is then translated into a timed-automata model which enables schedulability analysis as well as being able to verify that memory usage and power consumption are within certain limits. In the case where a system is not schedulable, the tool provides useful information, about what caused the missed deadline. We do not propose any particular methodology for design space exploration, but provide a tool, MoVES where embedded systems can be modelled and verified in the early stages in the design process. A key aspect is to provide a framework which allows system designers to explore alternatives in an easy and efficient manner, i.e. making it easy to change, update or even add new strategies and algorithms for task scheduling and allocation, and to perform changes in the execution platform by adding or removing components and interconnects.

⋆ This work was supported in part by ARTIST2 (IST-004527), MoDES (Danish Research Council 2106-05-0022) and DaNES (Danish National Advanced Technology Foundation).

The tool has been tested on a number of examples including a smart phone example, showing the ability to handle systems of realistic size.

2. RELATED WORK

Modelling and verification of embedded systems can be divided into simulation-based approaches and formal approaches. The simulation-based approaches such as ARTS, Madsen et al. [2004b], provide valuable feedback for the system designer in terms of understanding the overall behavior. The simulation-based approaches do, however, not give any guarantees and do not capture all critical cases that must be covered, in order to guarantee the absence of errors like missed deadlines and memory overflow.

Richter et al. [2003] propose a formal approach based on *event flow interfacing*. Looking at the timing properties of the input events of a subcomponent in the system, the output event flow is calculated for this subcomponent. This process is iterated along the data flow in the system. The iterative process will eventually find out if the system is schedulable or not.

Fersman et al. [2002] proposes a strategy for system level scheduling. This strategy is used in the TIMES tool. The tool is, however, limited to the single processor domain and the combination of dynamic scheduling algorithms and dependencies is not possible.

Medina et al. [2001] has constructed a tool, MAST, which uses classic scheduling theory in order to cover many scheduling problems and Thiele et al. [2000] provides a real-time calculus for scheduling of preemptive tasks. These approaches covers multiple processing elements but are limited to static priority scheduling.

Pop et al. [2000] proposes an approach for schedulability analysis of processes and message scheduling. This approach takes the administrative overhead of switching into account and uses a clock driven scheduling for the bus, while processing elements are scheduled according to fixed priority preemptive scheduling.

Angelov et al. [2006] proposes in the framework COMDES-II, which handles distributed embedded actors that communicate with each other by exchanging signals. Tasks are executed in a preemptive priority-driven scheduling environment, where tasks are released by a triggering event, which could be a periodic timing event. The output from the tasks are sent when the deadline arrives, which means that the system will always behave in worst case. The tasks interacts directly, without any knowledge of the execution platform. Furthermore the verification is based on schedulability analysis, which can not take the multiprocessor paradigm and shared resources into account.

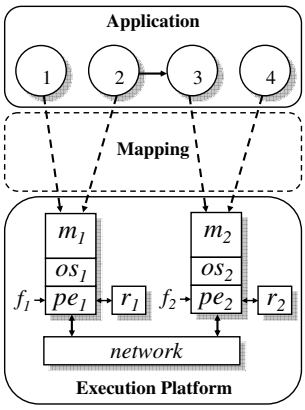
None of the above approaches, capture the exact behavior of an embedded system at a system-level in a formal way.

Describing the interaction of all sub-parts of the embedded system model formally, allows for verification, that all possible states of the system holds some property (e.g. no missed deadlines) through model checking. This approach has been used by Brekling [2006] where it is shown, that it is possible to model an embedded system in a formal way, using timed-automata in UPPAAL. The proposed model used in this work has the same modular structure as the one originally proposed by ARTS.

3. EMBEDDED SYSTEMS MODEL

A system-level model of an embedded system can be described as a layered structure consisting of three different parts. Figure 1 illustrate these layers for a very simple example of an embedded system, which will be used to explain aspects of the model throughout the paper.

- The application is described by a collection of communicating sequential tasks. Each task is characterized by four timing properties, described later. The dependencies between tasks are captured by an acyclic directed graph, which might not be fully connected .
- The execution platform consists of several processing elements of possibly different types and clock frequencies. Each processing element will run its own real-time operating system, scheduling tasks in a priority driven manner (static or dynamic), according to their priorities, dependencies and resource usage. When a task needs to communicate with a task on another processing element, it uses a network. The set up of the network between processing elements must also be specified, and is part of the platform.
- The mapping between the application and the execution platform (shown as dashed arrows in the figure) is done by placing each task on a specific processing element. In our model, this mapping is static, and tasks can not migrate during run-time.



a)

task			
τ_1	4	4	0
τ_2	6	6	0
τ_3	6	6	0
τ_4	6	6	4

b)

task	*e*	*sm*	*dm*	*pw*
τ_1	2	1	3	2
τ_2	1	1	7	3
τ_3	2	1	9	3
τ_4	3	1	6	4
τ_m	1			

c)

	pe_1	pe_2
f_1	1	1
scheduling	EDF	EDF
allocation	PRI_INH	PRI_INH

Fig. 1. System-level model of an embedded system. Characterization of a) tasks, b) tasks on processing elements, and c) processing elements.

The top level of the embedded system consists of an application mapped on to an execution platform. This mapping is depicted in Figure 1 with dashed arrows. The timing properties in the figure originates from Sun and Liu [1996], while the memory and power figures are created for the purpose of demonstrating parameters of an embedded system.

3.1 Application Model

The application is modelled as a set of independent programs which are executed on the execution platform. Each

program is modelled as a task graph, i.e. a directed acyclic graph of tasks where edges indicate causal dependencies. A dependency from τ_2 to τ_3 means that τ_2 must finish before τ_3 can start. Dependencies are shown with solid arrows in Figure 1. A task is a piece of sequential code and is considered to be an atomic unit for scheduling. A task τ_j is periodic and characterized by a period (π_j), a deadline (δ_j), an offset (ω_j), and a fixed priority (fp_j) (used when operating system uses fixed priority scheduling). The properties of periodic tasks can be seen in Figure 1a) and are all given in seconds.

3.2 Execution Platform Model

The execution platform is a heterogeneous system, in which a number of processing elements (pe) are connected through a network.

Processing Element Model: A processing element pe_i is characterized by a *clock frequency* (f_i), a *local memory* (m_i) with a bounded size, and a *real-time operating system* (os_i). The operating system handles synchronization of tasks according to their dependencies using direct synchronization, Sun and Liu [1996].

Access to a shared resource r_m (such as a shared memory or a bus) is handled using a resource allocation protocol, which in the current version consist of one of the following protocols: Preemptive Critical Section, Non-Preemptive Critical Section or Priority Inheritance. The tasks are in the current version scheduled using either Rate Monotonic, Deadline Monotonic, Fixed Priority or Earliest Deadline First scheduling, Liu [2000]. The properties of a processing element can be seen in Figure 1c.

Network Model: Inter-processor communication takes place when two tasks with a dependency are mapped to different processing elements. In this case, the data to be transferred is modelled as a message task τ_m. Message tasks have to be transferred across the network between the processing elements. A network is modelled in the same way as a processing element. So far, only busses have been implemented in our model, however, Madsen et al. [2004a] have shown how more complicated inter-communication structures such as a mesh or torus network can be modelled. As a bus transfer is non-preemptable, message tasks are modelled as run-to-completion. This is achieved by having all message tasks running on the bus, i.e. the processing elements emulating the bus, using the same resource r_m thereby preventing preemption of any message task.

Intra-processor communication is assumed to be included in the execution time of the two communicating tasks, and is therefore modelled without the use of message tasks.

3.3 Mapping

A mapping is a static mapping of tasks to processing elements of the execution platform. This is shown as the dotted lines in Figure 1. The execution time (e_{ij}) measured in cycles, memory footprint (*static memory* (sm_{ij}) and *dynamic memory* (dm_{ij})) and power consumption (pw_{ij}) of a task τ_j, depends on the characteristics of processing element pe_i executing the task, and can be seen in Figure 1b. In particular, when selecting the operation frequency (f_j) of the processing element pe_i, the execution time in seconds, ϵ_{ij} of task τ_j can be calculated as, $\epsilon_{ij} = e_{ij} \cdot \frac{1}{f_i}$ To fully explore different mappings, each task has to be characterized on all processing elements which can execute it, i.e. it may be that a particular task can only be executed on a subset of the processing elements.

3.4 Memory and Power Model

In order to be able to verify that memory and power consumption stays within given bounds, the model keeps track of the memory and power costs in each cycle. Additional cost models can easily be added to the model as long as the cost can be expressed in terms of cost of being in a certain state.

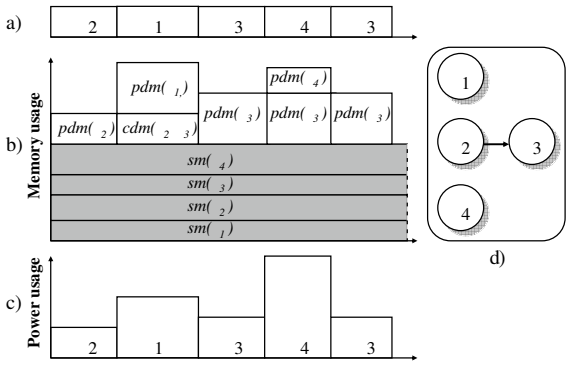


Fig. 2. Memory and power profile for pe_1 when all four tasks in Figure 1 are mapped onto pe_1. a) schedule where τ_3 is preempted by τ_4. b) memory usage on pe_1, Static memory: sm, private data memory: pdm, and communication data memory: cdm. c) power usage. d) task graph from Figure 1

The memory model, includes both static memory allocation (sm), due to program memory, and dynamic memory allocation (dm), due to data memory of the task. The example in Figure 2a illustrates the memory model. It shows the scheduling and resulting memory profile (split into static and dynamic memory). The dynamic part is split into private data memory (pdm) needed while executing the task, and communication data memory (cdm) needed to store data exchanged between tasks. The memory needed for data exchange between τ_1 and τ_3 must be allocated until it has been read by τ_3 at the start of τ_3's execution. When τ_3 becomes preempted, the private data memory of the task is still allocated until the task finishes.

Currently, a very simple approach for the modelling of power has been taken. When a task is running, it uses power (pw). The power usage of a task is zero at all other times. The possible different power usages of tasks can be seen as the heights of the execution boxes in Figure 2c.

4. TIMED-AUTOMATA MODEL

The embedded system model has been formalized using timed-automata semantics of UPPAAL. The structure of

the UPPAAL semantics resembles the structure of the model described in the previous section. The UPPAAL model consist of a number of timed automata, each representing a component of the embedded system model. The composition can formally be described as follows:

$$System = ExecutionPlatform \parallel Application$$
$$ExecutionPlatform = \parallel_{i=1}^{m} pe_i \parallel_{k=1}^{l} pe_k$$
$$pe_i = Controller_i \parallel Synchronizer_i \parallel Allocator_i \parallel Scheduler_i$$
$$Application = \parallel_{j=1}^{n} \tau_j$$

where $\parallel$ means parallel composition of timed automata in UPPAAL, m is the number of processing elements, l is the number of busses and n is the number of tasks.

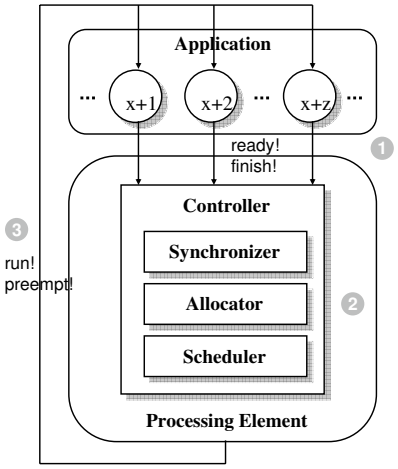


Fig. 3. Interaction of the z tasks τ_{x+1} to τ_{x+z} with the single processing element to which they are mapped.

4.1 Execution Platform

The formal description of the platform consists of parallel composition of a number of processing elements. Each of these consists of a Controller, Synchronizer, Allocator and Scheduler. The Controller receives `ready` or `finish` signals from the tasks of the application which are mapped to the processing element (see 1 in Figure 3), activates synchronization, allocation and scheduling to find the task with highest priority (see 2 in Figure 3), and finally sends `run` or `preempt` signals back to the tasks (see 3 in Figure 3).

The controller represents the operating system and its timed automata model can be divided into three parts. In the first part, input signals (`ready` and `finish`) are received. In the middle part, the actual operating system is run, synchronizing, allocating and scheduling. In the last part, output signals (`run` and `preempt`) are issued. These three parts correspond to the three steps in Figure 3.

4.2 Application and Tasks

The formal description of the application consists of a parallel composition of tasks. Tasks are receiving `run` or `preempt` signals from the operating system (i.e. the controller) and are producing the `ready` or `finish` signals to the operating system. Figure 4 shows a simplified version of the automata for a periodic task. The states

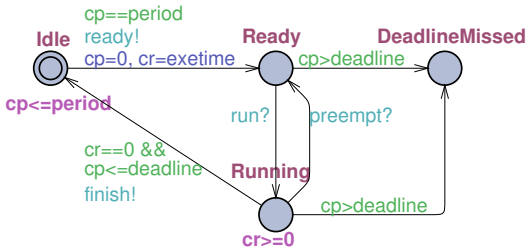


Fig. 4. Simplified task automata.

of a task moves between being ready, running, preempted (same as ready), and idle waiting for a new period to start or for an offset of the first task execution. The full task model consist of additional 6 vertices and 11 transitions which basically handles the task initialization (e.g. possible offset) and to be able to start and stop the task execution which requires discrete time intervals. The `cp` and `cr` in Figure 4 are clocks used to determine how far into its period the task is (`cp`) and how long the task has to run before it has finished execution (`cr`).

5. USING THE MODEL

In order to make the model usable for system designers, details of the timed-automata model is encapsulated in a tool called MoVES , which is meant to assist in designing, verifying and reconfiguring embedded systems in an intelligent way. The system designer needs to have an understanding of the embedded-system model, but not necessarily of the timed-automata model. It is assumed that tasks and their timing properties etc, are already defined and therefore MoVES is only concerned with helping the system designer to configuring the execution platform and perform the mapping of tasks on it.

The timed-automata model is created from a textual description which resembles the embedded system model presented in section 3. For easy prototyping, the textual description is currently embedded in Java (see Appendix A). MoVES uses UPPAAL as back-end to analyze the user's model and to verify properties of the embedded system through model-checking. UPPAAL can produce a diagnostic trace following verification. MoVES transforms this trace into a task schedule shown as a gantt chart. Currently this schedule is shown as a textual representation.

5.1 Example

To illustrate the design and verification process, consider the embedded system from Figure 1.

Through the UPPAAL verification MoVES is able to present the results to the user. Results of verifying the system from Figure 1 with both pe_1 and pe_2 using rate monotonic scheduling can be seen below:

```
E<>missedDeadline: true
E<>allFinish(): false
E<>totalCostInSystem(Power) == 7:true
E<>totalCostInSystem(Power) > 7: false
E<>costOnPE[0][Memory] == 17: true
E<>costOnPE[0][Memory] > 17: false
E<>costOnPE[1][Memory] == 12: true
E<>costOnPE[1][Memory] > 12: false
```

The verification results show several properties of the system. Firstly, the system cannot be scheduled in the given form i.e. it misses a deadline. Secondly, at no point does the system use more than 7 units of power, but at some point before missing the deadline 7 units of power is used. Finally, in regard to memory usage it is verified that pe_1 uses 17 units of memory at some point before missing the deadline but not more, and pe_2 uses 12 units but not more.

The trace produced by verification of the system in MoVES is shown here:

```
            5   10
Task: 1 1100110011
Task: 2 0010001000
Task: 3 0000110011
Task: 4 ----001100X
Task: 5 0001000100
```

The notation of the schedule is: 0 for idle, 1 for running, - for offset and X for missed deadline. It is shown that task 4 misses a deadline after 11 execution cycles. Note that task 5 is the message task between task 2 and 3.

In order to examine what can be done to improve the system, we attempt verification of the same system where p_2 uses earliest deadline first scheduling. The verification results can be seen below:

```
E<>missedDeadline: false
E<>allFinish(): true
E<>totalCostInSystem(Power) == 7: true
E<>totalCostInSystem(Power) > 7: false
E<>costOnPE[0][Memory] == 11: true
E<>costOnPE[0][Memory] > 11: false
E<>costOnPE[1][Memory] == 12: true
E<>costOnPE[1][Memory] > 12: false
```

Again we can analyze the results. Firstly this system is now schedulable, as can be seen by the `E<>allFinish()` query being true. The system still has the same power usage properties as with rate monotonic scheduling used on pe_2 but the verification shows that at no point will the revised system - i.e. where pe_2 uses earliest deadline first - use more than 11 units of memory, recall that the system where pe_2 used rate monotonic scheduling already before missing a deadline had at some point used 17 units of memory.

The trace produced by verification of the revised system in MoVES is shown here:

```
            5   10   15   20   25   30
Task: 1 1100110011001100110011001100110011
Task: 2 0010001000000010001000000001000
Task: 3 0000110001100000110001100000011
Task: 4 ----0011100111000011100111000
Task: 5 0001000100000001000100000000100
```

6. TOWARDS INDUSTRIAL APPLICATIONS

The MoVES tool has been evaluated using applications that are part of a smart phone, in order to show that the tool is applicable on a typical embedded system. The smart phone includes the following applications: GSM-encoder, GSM-decoder and MP3-decoder (103 tasks total), as seen in Figure 5. These applications do not together make up the complete functionality of a smart phone, but are used as an example, where the number of tasks, their dependencies and their timing properties are realistic. The applications and their properties in the smart phone example, originates from experiments of executing C-code on different processors done by M. Schmitz Schmitz et al. [2004]. The timing properties, period and deadline, of the tasks, are imposed by the application and can be seen in Table 1. The smart phone example has been verified using worst-case execution times only.

Table 1. Application and pe characteristics

Application	Tasks/ Edges	Deadline/ Period (s)
GSM Encoder	53/80	0.020
GSM Decoder	34/55	0.020
MP3 Decoder	16/15	0.025

pe	Frequency (MHz)
GPP0	25
GPP1	10
GPP2	6.6
FPGA	2.5
ASIC	2.5

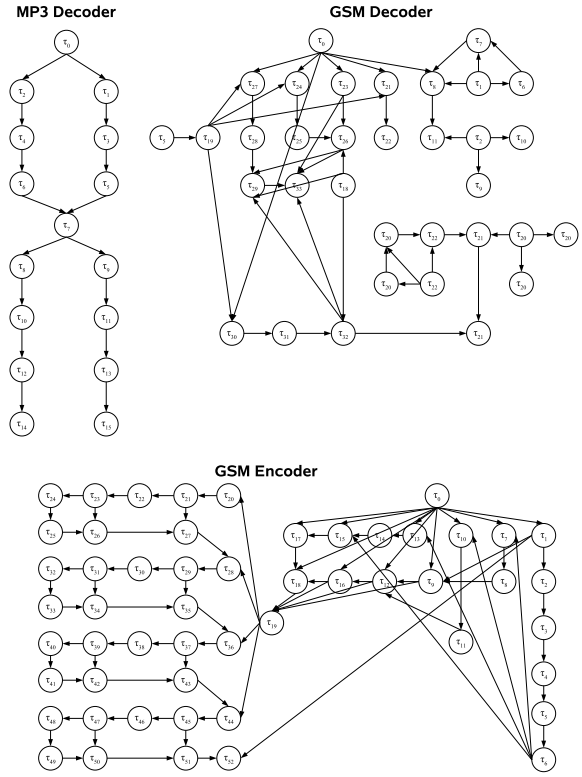


Fig. 5. Task graph for 3 applications from a smart phone taken from Schmitz et al. [2004].

The execution cycles, memory usage and power consumption of each task, depend on the processing element. These properties of the tasks, have been measured, by simulating the execution of each task on different kinds of processing elements (GPP, FPGA, ASIC) as seen in Table 1. The execution cycles range from 52 to 266687 and the periods range from 0.02 to 0.025 seconds giving a total number of 504 tasks to be executed in the hyper period of the system.

The three applications have been mapped onto four general purpose processing elements running at 25 MHz (GPP0) connected by a bus. The parallelism of the MP3-decoder has been used to split this application onto two processing elements. The two other applications run on their own processing element.

Having defined the embedded system with the application, execution platform and mapping described above, the MoVES tool is used to verify schedulability, maximum memory usage and power consumption. In this case the system is schedulable and the maximum memory usage and power consumption is 1500 bytes and 1000 mw. The verification of the smart phone example takes roughly 3 hours on a 64bit Linux server with AMD dual core processor with 2 GB of memory.

It is possible that better designs exist, e.g. where less power is used. A general purpose processor could for example run at lower frequency or be replaced by an FPGA or an ASIC. This is however not the focus of this case study. The output of our tool, can however be used as input for a tool which can optimize execution platform and mapping. We have shown that our tool is capable of verifying systems of realistic size.

7. CONCLUSION

This paper has presented a tool for formal verification of embedded multi-core systems, using model checking of timed automata in UPPAAL. Using model checking, properties such as schedulability, memory usage and power consumption can be analyzed. We have emphasized the

Emphasis has been on explaining the general concepts of the system-level embedded system model supporting applications executing on a multi-core execution platform and how to use the model for design exploration of embedded systems design. Hence, the timed-automata model has only been briefly outlined. In verification of the smart phone example a key challenge has been dealing with state space explosion. The successful verification is an important milestone in being able to handle industrial sized applications.

REFERENCES

Christo Angelov, Xu Ke, and Krzysztof Sierszecki. A component-based framework for distributed control systems. In *EUROMICRO '06: Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 20–27. IEEE Computer Society, 2006.

Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

A. Brekling. Modelling and verification of mpsoc. Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, 2006.

Elena Fersman, Paul Pettersson, and Wang Yi. Timed automata with asynchronous processes: Schedulability and decidability. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 67–82, 2002.

Jane W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

J. Madsen, S. Mahadevan, and K. Virk. Network-centric system-level model for multiprocessor soc simulation. In J. Nurmi, H. Tenhunen, J. Isoaho, and A. Jantsch, editors, *Interconnect-Centric Design for Advanced SoC and NoC*, chapter 13, pages 341–365. Kluwer Academic Publishers / Springer Publishers, July 2004a.

J. Madsen, K. Virk, and M. J. Gonzalez. A systemc-based abstract real-time operating system model for multiprocessor system-on-chip. In *Multiprocessor System-on-Chip*. Morgan Kaufmann, 2004b.

Julio L. Medina, Michael Gonzlez Harbour, and Jos M. Drake. Mast real-time view: A graphic uml tool for modeling object-oriented real-time systems. pages 245–256, 2001.

Paul Pop, Petru Eles, and Zebo Peng. Bus access optimization for distributed embedded systems based on schedulability analysis. In *DATE '00: Proceedings of the conference on Design, automation and test in Europe*, pages 567–575. ACM Press, 2000.

Kai Richter, Marek Jersak, and Rolf Ernst. A formal approach to mpsoc performance verification. *Computer*, 36(4):60–67, 2003.

Marcus T. Schmitz, Bashir M. Al-Hashimi, and Petru Eles. *System-Level Design Techniques for Energy-Efficient Embedded Systems*. Kluwer Academic Publishers, 2004.

Jun Sun and Jane W.-S. Liu. Synchronization protocols in distributed real-time systems. In *International Conference on Distributed Computing Systems*, pages 38–45, 1996.

L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systeme. In *IEEE Internation Conference on Circuits and Systems*, 2000.

Appendix A. SOURCE CODE OF EXAMPLE

```
public class MPSoC {
  public Application apps;
  public Platform pl;
  public MPSoC(int granularity) {
    //Defines tasks(bcet,wcet,Deadline,offset,period,FP)
    Task t1 = new Task(2, 2, 4, 0, 4, 1);
    Task t2 = new Task(1, 1, 6, 0, 6, 2);
    Task t3 = new Task(2, 2, 6, 0, 6, 3);
    Task t4 = new Task(3, 3, 6, 4, 6, 4);
    Task tm = new Task(1, 1, 6, 0, 6, 5);

    //Defines processors
    Processor p1 = new Processor(1, RM, PRI_INH);
    Processor p2 = new Processor(1, RM, PRI_INH);
    Processor pm = new Processor(1, RM, PRI_INH);
    //Assigns tasks to processors
    Task[][] tasks = {{t1,t2},{t3,t4},{tm}};

    //Adds the processors to the system
    Processor[] ps = {p1,p2,pm};
    Resource bus = new Resource();

    Cost memory = new Cost(tasks);
    Cost power = new Cost(tasks);
    Cost[] ca = {memory, power};

    pl = new Platform(ps);
    apps = new Application(tasks, ca, granularity);
    apps.useResource(tm, bus);

    memory.set(t1,1,0,0,3,3);
    memory.set(t2,1,0,0,7,7);
    memory.set(t3,1,0,0,9,9);
    memory.set(t4,1,0,0,6,6);

    power.set(t1,0,0,0,2,0);
    power.set(t2,0,0,0,3,0);
    power.set(t3,0,0,0,3,0);
    power.set(t4,0,0,0,4,0);

    memory.share(t2,t3,7);

    //Adds dependencies to the system.
    apps.addDep(t2,tm);
    apps.addDep(tm,t3);
  }
}
```