MoVES - A tool for Modelling and Verification of Embedded Systems

Jens Ellebæk Nielsen and Kristian Staalø Knudsen

Kongens Lyngby 2007
IMM-MASTER-2007-43
April 30, 2007

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Abstract

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 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-processor System-on-Chip (MPSoC) designs, integrating multiple processing elements connected through an on-chip network. One or more applications are divided among these processing elements. As these systems become more complex, the interaction between hardware and software becomes more unpredictable and problems such as memory overflow, data loss and missed deadlines are more likely to occur. System-level verification, of schedulability and resource usage, has therefore become one of the most relevant fields of study in designing real-time systems.

This thesis proposes a model of MPSoCs, where the interaction of all subcomponents of the MPSoC is captured in an exact formal way. This allows for formal verification using model-checking of properties such as schedulability and resource usage. The model is implemented using timed-automata in UPPAAL [3].

A tool is built, on top of the timed-automata model, which allows designers of embedded systems to analyze MPSoC systems early in the design phase.

The timed-automata model has been evaluated using a number of synthetic applications in order to demonstrate correct behavior. It has also been shown that the tool is able to handle real-life applications from a smart phone, with more than 100 tasks.

Resumé

Indlejrede computer systemer benyttes i flere og flere apparater, fra husholdnings maskiner til mobiltelefoner, PDA'er og biler. Mange af disse systemer har begrænsede ressourcer som hukommelse og strøm til rådighed. Herudover skal de udføres til en bestemt tid, afhængigt af deres anvendelse.

Da det bliver sværere at øge en computers effektivitet ved at benytte sekventiel udførelse, rykker tendensen over imod at benytte Multi-processor System-on-Chip (MPSoC) designs, hvor flere processorer integreres og forbindes af et on-chip netværk. En eller flere applikationer er fordelt på de forskellige processerings elementer. Da disse systemer bliver mere komplekse, bliver samspillet mellem software og hardware mere upålidelig, og problemer som tab af data og overskridelse af tidsfrister optræder hyppigere. System-niveau verifikation af korrekt udførelse og ressource forbrug, er derfor blevet et af de mest relevante forskningsområder i forbindelse med design af tidstro systemer.

Denne afhandling foreslår en model af MPSoCs, hvor samspillet imellem alle delkomponenter er behandlet på en eksakt formel måde. Dette giver mulighed for at benytte formel verifikation ved model-testning af egenskaber som: planmæssig udførelse og ressource forbrug. Modellen er implementeret ved brug af tidsautomater i UPPAAL [3].

Der er blevet bygget et værktøj oven på tidsautomat modellen, som giver designeren af indlejrede systemer mulighed for at analysere MPSoC systemer allerede tidligt i design fasen.

Tidsautomat modellen er blevet testet ved brug af en række syntetiske applikationer, for at kontrollere korrekt opførelse. Det er endvidere blevet vist at værktøjet er i stand til at håndtere virkelige applikationer fra en multimedie-telefon med over 100 tasks.

Preface

This thesis was prepared at Informatics and Mathematical Modelling, at the Technical University of Denmark in partial fulfillment of the requirements for acquiring the M.Sc. degree in engineering.

The thesis deals with formal modelling and verification of Multi-processor Systems-on-Chip (MPSoC), using timed automata in UPPAAL. The thesis also focusses on creating a tool, by which designers without knowledge of UPPAAL are able to create and verify MPSoC systems. The work described in this thesis is closely related to the MoDES[1] and DaNES[2] projects.

The project was completed in the period from November 1st, 2006 to April 30th, 2007 under the supervision of Associate Professor Michael R. Hansen and Professor Jan Madsen.

An abstract containing the major findings from this project was submitted and accepted for a University Booth presentation at the DATE 07 conference in Nice in April 2007 [8]. A short lecture was given, at the MoDES workshop at SDU, Sønderborg in March.

Jens Sten Ellebæk Nielsen   Kristian Staalø Knudsen

Lyngby, April 2007

[1]MoDES - Model Driven Development of Intelligent Enbedded Systems, funded by the Danish Council for Strategic Research

[2]DaNES - Danish Network for Intelligent Embedded Systems, funded by "Højteknologi-fonden".

Acknowledgements

We would like to thank our supervisors Michael R. Hansen and Jan Madsen for their great support throughout the entire phase of our project. You have always taken the time to address our questions and have provided us with inspiring ideas and challenges.

We would also like to thank Jesper Knudsen, Thomas Korsgaard, Kåre Poulsen and Dan Søndergaard for their ideas and proof-reading.

Thanks to Jacob Illum at Aalborg University for spending time modifying the Verifyta program to accommodate our needs.

A special thanks goes to Aske Brekling, for his endless support and time spend on discussing unclear issues.

Contents

Abstract i

Resumé iii

Preface v

Acknowledgements vii

1 Introduction 1

1.1 Introduction to MPSoC 2

1.2 Related Work 3

1.3 Objective 5

1.4 Structure of this Thesis 6

2 MPSoC model 9

2.1 Application 11

2.2 Execution Platform 16

3 Timed-Automata Semantics **27**

3.1 UPPAAL model 28

3.2 Environment 39

3.3 Cost 40

3.4 Model checking 41

4 MoVES tool **47**

4.1 Flow in MoVES 48

4.2 Structure of MoVES 53

5 Evaluation **55**

5.1 Testcases 56

5.2 Smart phone 69

5.3 Feasible Sizes 72

5.4 Comparison 74

5.5 Summary 78

6 Future Work **81**

6.1 MPSoC features 81

6.2 MoVES tool 85

7 Conclusion **87**

A Terms **89**

B Introduction to timed automata 91

B.1 Finite automata 91

B.2 Communicating finite automata 92

B.3 Extended finite automata 94

B.4 Timed finite automata 95

B.5 UPPAAL 96

B.6 Model checking 97

B.7 Priorities 99

B.8 Select 99

C Using MoVES 101

C.1 Defining a system for MoVES 101

C.2 Running MoVES 105

D Evaluation Results 109

E Java code 125

E.1 MoVES.java 125

E.2 Processor.java 127

E.3 Task.java 128

E.4 Environment.java 129

E.5 Application.java 130

E.6 Verifier.java 132

E.7 BuildTA.java 137

E.8 Parser.java 159

E.9 Platform.java 162

E.10 Resource.java 164

E.11 Cost.java 165

E.12 help.txt 166

E.13 MPSoC.java for Smart Phone 167

F Entire Uppaal model **173**

F.1 Global declarations 173

F.2 Task 176

F.3 Control 180

F.4 Synchronizer 182

F.5 Allocator 183

F.6 Scheduler 184

F.7 Rescheduler 185

F.8 Triggered 186

F.9 Environment 189

G Contents of the CD-rom **191**

Chapter 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-processor Systems-on-Chip(MPSoC) designs, integrating multiple processing units connected through an on-chip network. One or more application are divided among these processing elements. As these systems become more complex, the interaction between hardware and software becomes more unpredictable 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 systems independently, as a minor change at one layer can greatly influence other layers. System-level verification of schedulability, taking all layers into account, has therefore become one of the most relevant fields of study in designing real-time systems.

A typical example where MPSoCs are useful, and are in fact used, is a modern mobile phone. Modern mobile phones have several real-time systems running at the same time. Imagine the scenario where a person is talking while using the menu system of the phone, and wants to show an jpeg image- at the same time. In this scenario, there are numerous applications running, for handling encod-

ing and decoding of audio, handling inputs from the buttons on the phone, and decoding the jpeg image in order to show it on the screen. Errors in these applications are annoying and damaging to the companies reputation. Therefore the companies want to ensure, that their products are not to error prone. In the case of mobile phones, it is not possible to simply add more and faster hardware, to make sure everything works out fine, as this is costly. When designing a mobile phone, the designers have chosen some applications that should be implemented on a hardware platform. One interesting question could be: What is the minimal execution platform (cheapest), which will execute these applications without generating errors due to loss of deadlines?

A modern car has numerous different integrated computer systems, handling everything from stereo and automatic windows to ABS-brake system and airbags. The traditional way of handling safety systems, is simply to isolate each safety critical circuit from the rest of the electronic systems in the car. As more complex safety systems are implemented in the car, the need for a lot of independent systems arise. This is costly in hardware, as each system needs its own processing element. A vision is to integrate as many systems possible together using an MPSoC. This is however not acceptable to the car companies, because there is no way to guarantee, that everything works as intended, when systems become that complex. In order to integrate such circuits together, it must be verified that all sub systems work together in a proper timely manner.
The possibility of verifying timing properties is especially important in the second example, and verifying upper bounds for memory usage and power consumption is important for the first example. The next section shortly describes the way an MPSoC is modelled in this thesis.

1.1 Introduction to MPSoC

A system-level model design of an embedded system can be split in to three different parts as seen in Figure 1.1.

1: Application The application is described by a collection of communicating sequential tasks. Each task is characterized by an offset, deadline, period and execution time. The dependencies between tasks are captured by an acyclic directed graph.

2: Execution Platform A specific execution platform must be chosen. The execution platform consists of numerous processing elements of possibly different types and frequency. Processing elements can roughly be divided into three types: General Purpose Processors (GPP), Field Pro-

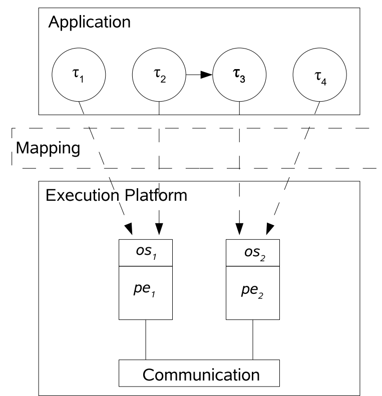

Figure 1.1: System level model of MPSoC

grammable Gate Array (FPGA) and Application Specific Integrated Circuits (ASIC). Each processing element will run its own real-time operating system, schedule tasks according to their priorities, dependencies and resource usage. When a task needs to communicate with a task on another processing element, it happens through an on-chip network. The set up of the network between processing elements must be also be specified and is part of the platform.

3: Application mapping The mapping between the application and the platform is done by placing each task on a specific processing element. This mapping is static.

A thorough description of how the application and the execution platform is modelled, is given in Chapter 2.

1.2 Related Work

Modelling and verification of MPSoC systems can be divided into simulation-based approaches and formal approaches. The strengths and shortcomings of each approach are shortly discussed.

The simulation-based approaches provides a valuable feedback for the MPSoC designer in terms of overall behavior. The simulation-based approaches does however not give any guarantees and does not capture all critical cases that

must be covered, in order to guarantee the absence of problems like missed deadlines and memory overflow. Further problems arise in the form of system anomalies due to multiple processors, shared resources and unpredictable execution times. An example is the simulation-based framework ARTS [13].

In order to handle the short comings of the simulation-based approaches several more or less formal approaches have been proposed. In [15], an approach for analyzing communication delays in message scheduling, together with optimization strategies for bus access, is presented.

Thiele et al. [21] provides a real-time calculus for scheduling of preemptive tasks. This approach assumes static priorities and does not capture the concept of timing anomalies where local best-case executions lead to global worst-case schedules.

Richter, Jersak and Ernst [18] propose a formal approach based on event flow interfacing. Looking at the 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 either the system is schedulable or not. It is however not clear if the process always terminates. The method also uses over-approximation and does not capture the exact execution of a MPSoC system.

In [9] a strategy for scheduling on a system level is proposed. 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.

M. Harbour has constructed a tool [14], which uses classic scheduling theory in order to cover many scheduling problems. This approach has its clear limitations for systems, with timing properties that are hard to model mathematically. As many embedded system are reactive in nature and have real-time requirements, it is often infeasible to analyze them statically at compile time. Also, none of the above approaches, capture the exact behavior of an MPSoC at a system level.

An other formal approach is model checking (checking all possible executions μ if they lead to the satisfaction of a property φ), describing the precise interaction of all sub-parts of the MPSoC system. In this way all possible states of the system can be checked for satisfying some property. This approach has been used by A. Brekling in [4, 5] where it is shown, that it is actually possible to model an MPSoC in a formal way, using timed-automata in UPPAAL. The proposed MPSoC used in this work has the same modular structure as the one originally proposed by ARTS which can be seen in Figure 1.2. A. Brekling

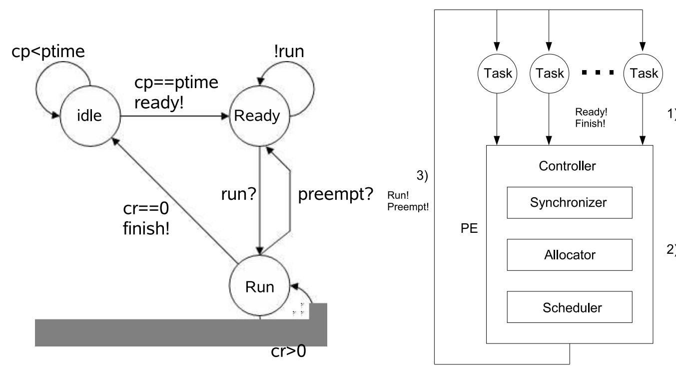

Figure 1.2: The ARTS task model and execution platform

modelled a small subset of features of an MPSoC system, including dynamic scheduling and inter-processor dependencies of periodic tasks. The feasible size of systems were 10 tasks on 6 processing elements. This work has opened up for a new way of verifying properties of embedded systems in general.

The precise objective of our thesis is defined in the next section.

1.3 Objective

The objective is to develop a formal timed-automata semantics for a system-level MPSoC using UPPAAL. The formal semantics gives the ability to model-check properties of timing, memory usage and power consumption. A formal semantic including some features of an MPSoC has been developed by A. Brekling [4]. Our project aims to develop a model comprising a larger set of features, making the model more realistic and useful to designers. Furthermore the size of systems, which can be verified, should be increased to a level where systems such as the smart phone described in Section 5.2 can be verified.

The new features included in this project are:

- Resource allocation as part of the real-time operating system.
- Non-periodic tasks (sporadic and aperiodic).
- Cost model for investigating memory usage and power consumption pr. processing elements and on the total execution platform.

- Deadline types (hard, firm, soft).
- Inter-processor communication via busses.
- Non-deterministic[1] execution times of tasks, in an interval between a best and worst case.
- Different clock frequencies on processing elements.

Using the formal model to model check MPSoC systems, we want to develop a tool which assist in the design process of MPSoCs. The tool should be usable to people who understand the concept of MPSoCs, but who do not necessarily have an understanding of timed automata.

The tool should primarily be able to answer the schedulability problem concerning MPSoCs, and at the same time be able to verify that memory usage and power consumption are always within a specified bound. In the case where a system is not schedulable, the tool should provide useful information, about what went wrong, to the designer, in order to reconfigure by changing the execution platform and mapping of tasks. It is noted that automatic reconfiguration of the platform and mapping of tasks, are not in the scope of this thesis, and are in fact an area of research by it self.

In short the three main points of this thesis is to:

- Comprise a larger set of features in the MPSoC domain.
- Optimize the model for efficient model-checking.
- Provide a tool which is usable for a MPSoC designer.

1.4 Structure of this Thesis

This thesis is structured as follows. The next chapter defines our exact understanding of an MPSoC system. The desired behavior is presented and unclear concepts are analyzed in detail.

In Chapter 3, the behavior of the MPSoC system is implemented as a formal semantics using timed-automata in UPPAAL.

[1] The execution times of sequential code is not in fact non-deterministic, but depends on the input variables. However at the task level, the input variables are abstracted away and the execution times are modelled as non-deterministic.

Having developed the timed-automata model, the tool, for assisting in designing MPSoC systems, is build upon the timed-automata model and is described in Chapter 4. This chapter can be read without knowledge of the timed-automata model.

The evaluation of both the timed-automata model and the tool is found in Chapter 5. This chapter contains functional testing, a case study of a real world example and an evaluation of size of systems, which can feasibly be verified.

A discussion on further development is found in Chapter 6, and a general conclusion is given in Chapter 7.

All terms in relation to MPSoCs can be found in Appendix A. Appendix B contains an introduction to timed automata, originally enclosed in [4]. A user guide for the tool is provided in Appendix C. The rest of the appendices contains test output, source code and the entire UPPAAL semantics.

In addition to this thesis a CD-ROM is enclosed. The CD-ROM contains the abstract, poster and handout which was presented at the DATE'07 conference. The CD-ROM also contains the slides for the presentation at the MoDES workshop in April 07. Furthermore the CD-ROM contains the source code for the examples, the source code for the tool (with an API in HTML) and the thesis in an electronic version.

It contains the source code for the developed tool, the examples and also the thesis in an electronic version.

Chapter 2

MPSoC model

This chapter describes how the generic MPSoC system level is modelled. The model is inspired by ARTS [13] and exhibit the same modular structure. A top-down approach will be taken.

A formal description of the components, which make up the MPSoC model, can be expressed as follows:

$$
\begin{aligned}
&System = Application \parallel Execution\ Platform\\
&Application = \parallel_{j=1}^{n} \tau_j\\
&Execution\ Platform = \parallel_{i=1}^{m} pe_i
\end{aligned}
$$

Where τ_j is the j'th of n tasks and pe_i is the i'th of m processing elements and $\parallel$ means parallel composition of components. A description of all terms and notation can be found in Appendix A.

The top level of the MPSoC consists of an application mapped on a platform. This is depicted in figure 2.1 with dashed arrows. All numbers in this figure are made up, in order to demonstrate the parameters of a MPSoC system. This case only exemplifies periodic tasks.

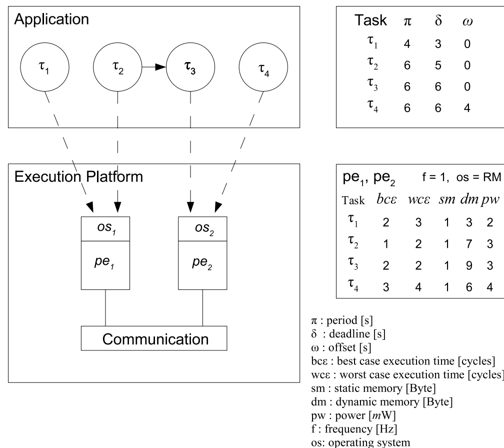

Figure 2.1: MPSoC system. See text for explanation.

The application specification is modelled as a set of independent programs which have to be executed on the execution platform. As mentioned, 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 starts. This is also written: $\tau_2 \prec \tau_3$. Dependencies are shown with solid arrows in figure 2.1. A task is a piece of sequential code and can not be split on to multiple processors. A task τ_j is either periodic and represented by a period (π), a deadline (δ), an offset (ω), given in seconds, and a fixed priority (fp) (used when operating system uses fixed priority scheduling) or non-periodic with a deadline and an trigger event channel. The deadline of tasks can be hard, firm or soft as later described in Section 2.1.2. Non-periodic tasks can either be triggered internally by another task, or externally by the environment. The properties of periodic tasks are seen on the top right side in figure 2.1 and are all given in seconds. The best and worst case execution times ($bc\epsilon$), ($wc\epsilon$), memory footprint (*static memory* (sm) and *dynamic memory* (dm)) and power consumption (pw) of the task will depend on the characteristics of the processing element to execute it.

The execution platform is a heterogeneous MPSoC in which a number of processing elements (pe) are connected through an on-chip network. 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) which schedules the tasks mapped to the processing element, handles resource allocation (such as

a bus or a shared memory) and inter-task dependencies. The properties of the types of processing elements can be seen in the lower right rectangle in Figure 2.1

An application implementation is a static mapping of tasks to processing elements of the execution platform. This is shown as the dotted lines in figure 2.1. When a task τ_j is mapped to a processing element pe_i ($\tau_j \mapsto pe_i$), we get a worst case execution time ($wc\epsilon_{ij}$) and a best case execution time ($bc\epsilon_{ij}$). This means that a task may complete its execution anywhere in the interval between $bc\epsilon_{ij}$ and $wc\epsilon_{ij}$. The exact values of $bc\epsilon_{ij}$ and $wc\epsilon_{ij}$, given in execution cycles, are dependent on the architecture of the processing element. Once a task is mapped onto a processing element, the properties period (p), deadline (d) and offset (o) in execution cycles(not time) are calculated as described in Section 2.2.6.

The following three sections explain how the layers of application, execution platform and communication are modelled. In order to keep a strict separation of the application and the execution platform, the timing properties of a task, δ, ω *and* π , are given in seconds and are imposed by the application[1]. The execution cycles of a task on the other hand, depends on the architecture and instruction set of the processing element on which it is executed. The relationship between timing properties in seconds and the execution time in cycles is dependent on the processor frequency and is explained in the fourth section.

2.1 Application

As mentioned the application consist of tasks. First the periodic-task model with hard deadline is described. Secondly the modelling of types of deadlines, (hard, firm and soft) are described. Finally non-periodic tasks are described.

Each task in the application is modelled as a finite state machine with the states Idle, Ready and Running. The state machine is illustrated in Figure 2.2. Tasks moving between these states, interacts with the operating system, and hence the moving is done in zero time.

[1]The timing properties of tasks was originally modelled in execution cycles by A. Brekling, making an implicit assumption about the processor frequency on which the task runs.

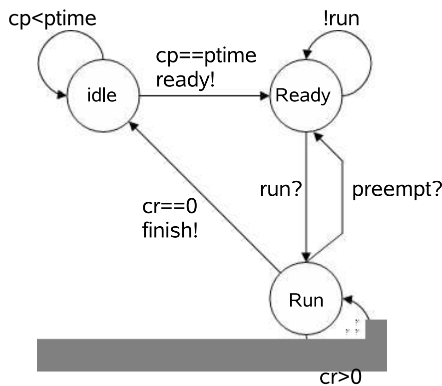

Figure 2.2: Finite State Machine with 3 states for a single task

2.1.1 How the task model works

In each clock cycle, all tasks takes a loop transition to its current states, in order to update variables associated with time. *cp* measures the time since the period started, and is incremented by one pr. time unit elapsed. *cr* measures the remaining execution time and is only decremented in the Running state. When a task becomes ready, *cr* is set to a random value in the interval between best and worst case execution times.

When a task is instantiated, it becomes ready if it has no offset. Otherwise it waits in the Idle state until the offset is exceeded. When the task becomes ready, it issues a ready signal to its processing element and moves to the Ready state. The task waits in the Ready state, until it receives a run signal from its processing element. When a run signal is received, the task moves to the Run state. In the Run state the task can either receive a preempt signal, because a higher priority task has become ready (or has resolved all its dependencies), or the task finishes and sends a finish signal. In the case of a preempt signal, the task goes back to ready and waits to be scheduled again. In the case of a finish signal, the task moves to the Idle state and waits for the next period to start. In this model, the task does not know anything about other tasks or processing elements. It can only send signals to its own processing element, when it is ready or finished and react to run and preempt signals from its processing element.

The following inequality is always true if the task has not missed its deadline: $cp \leq d$. In other words: When the inequality becomes false the task has missed a deadline. If the equality is not satisfied, the flag *missedDeadline* is

set to true. If all tasks meet their deadlines, the flag *missedDeadline* is always false. Timing is one of the properties that can be verified using this model. Verification of timing along with other properties are described in Section 3.4

2.1.2 Types of deadlines

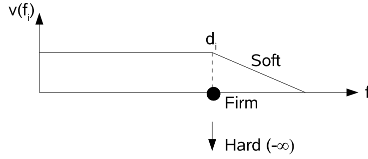

Figure 2.3: The value before, at and after the deadline of the three types of tasks.

In a real-time systems the concept of value, V, of a schedule is introduced in order for the operating system to make as good scheduling decisions as possible. The *value*(v) of a task is dependent on its finish time [6]. The total value, V of the particular schedule, is the sum of the value function, $v(f_j)$ of the finish time f_i for task τ_j:

$$V = \sum_{j=1}^{n} v(f_j)$$

where n is the total number of tasks on all processing elements.

Tasks which still have some value after its deadline are said to be *soft*. The value of a task with a soft deadline, decreases with time after the deadline. Tasks with hard deadlines have a value of minus infinity after its deadline, thereby modelling that a miss is not acceptable. A task with a firm deadline type is somewhat in between. A task may be real-time and add zero value to the system if completed after its deadline without jeopardizing the entire system. The three types of tasks can be seen in Figure 2.3. Based on these definitions we can decide how to model misses of the different types of deadlines. In [6], there are many so called overload management strategies for optimizing real-time systems including tasks with soft and firm deadlines. We will however use a simple approach in order to show the possibility of including an overload management strategy in our model. The type of deadline is defined for each task in the application specification

It is critical, if a task with a hard deadline misses, and it makes no sense to

keep the system running. The system should go into a state where this can be detected.

If a soft deadline is missed, there is still some value in completing the task. The scheduling is continued as normal. The task which missed a soft deadline, will still be scheduled using whatever scheduling algorithm running on its processing element. In this case, the period of a task with soft deadline will become skewed compared to the rest of the system.

The third type of missed deadline is more interesting. The miss is not critical to the system, but there is no value in finishing the task. In this case the task is stopped from executing in this period and releases the processing element on which it was executing. The task goes back to being idle, waiting for the next period to start. This saves processing time equal to the remaining execution time of the task with a firm deadline. This time can now be used for other tasks which adds value to the schedule.

2.1.3 Non-periodic tasks

So far only periodic tasks have been discussed. In order to give a more realistic model, we want to model non-periodic tasks too. Non-periodic tasks can model events that occur in a non predictable way, or events that occur without a fixed period. Examples of such are user input of a system. In a mobile phone this could be the activation of buttons by the user, which are pressed at random times. According to [11] there are two types of non-periodic tasks. The *sporadic* tasks which have a hard deadline and the *aperiodic* tasks which have a soft deadline. Using the three types of deadlines, described in the previous section, it is possible to model non-periodic tasks with firm deadline. If a firm deadline is missed the system will continue and the execution of the task will stop, since there is no point in continuing (Recall that, a task with firm deadline adds no value to the system after its deadline).

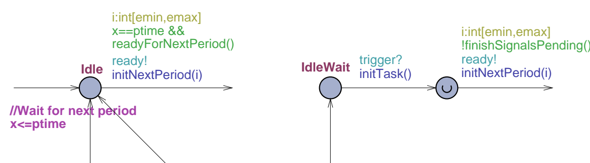

Figure 2.4: The Idle states from a normal task and a trigger task

Instead of becoming ready at the start of the period, a non periodic task becomes ready when it is triggered by an internal or external event. This is done by giving a non-periodic task a trigger channel. The internal triggering is done by the operating system of the task. When the triggering task finishes, the operating system sends a trigger signal to the task, making it move from its Idle state to the Ready state. This is the only main difference from periodic tasks. In Figure 2.4 the difference is seen in the guard at the transition between the Idle state and the Ready state. A non-periodic task can by nature not have an offset.

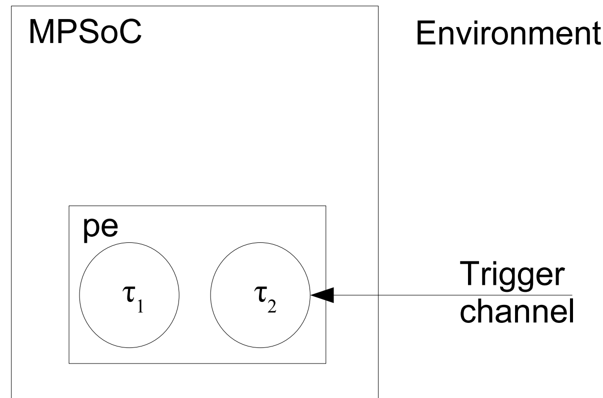

Figure 2.5: The interface to the MPSoC is the triggering channel of the non-periodic task τ_2

The external triggering of a task is done by defining an environment of the system, which can send a trigger signal to the task. In this way it is possible to define the interaction between the MPSoC and the environment through triggering events of non-periodic tasks. The modular structure of our MPSoC model, makes it possible to define and interchange the environments of the MPSoC dynamically. The interface between the MPSoC and the environment are the triggering channels of the non-periodic tasks. The interaction between the MPSoC and the environment is shown in Figure 2.5. In Section 3.2 the semantics of a simple environment, where a task is triggered with a minimum inter-arrival time is given.

2.2 Execution Platform

On a real processing element, execution of instructions are carried out in clock cycles. In this model of the execution platform, time is divided into clock cycles as well and each task takes a number of clock cycles to complete.

Between clock cycles, the operating systems only runs, if there is any change(finish or ready signals), and schedules a task to run in the next clock cycle. If there is no ready or finish signals, the operating system will not run, since it uses priority driven scheduling. The overhead time for the operating systems on the execution platform is assumed be zero. This is a general assumption in modelling real-time systems, because the overhead time often is negligible compared to the task execution times.

The processing elements have the possibility of communicating in each clock cycle, in this way ensuring a correct global schedule, because the operating systems can exchange information about which dependencies has been resolved. The real-time operating system consists of three parts as seen in Figure 2.6.

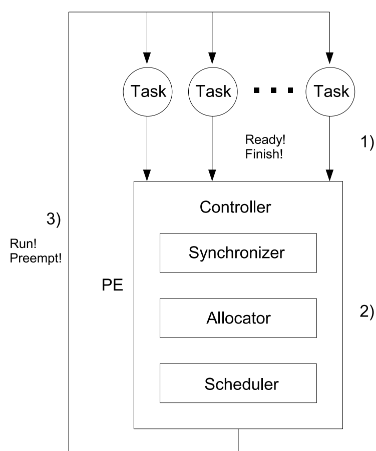

Figure 2.6: The execution platform consisting of synchronizer, allocator, scheduler

Synchronizer The synchronizer takes care of dependencies between tasks. Only ready tasks with no unresolved dependencies become synchronized.

Allocator The Allocator makes sure that all resources needed by a task is avail-

able. Among the synchronized tasks, only tasks with available resources are allocated.

Scheduler The scheduler chooses the task with highest priority to run, among the allocated task on its processing element, according to its scheduling algorithm.

In order to make it easier to model the operating systems both internally and their interaction, a controller is introduced as a top level component(on each processing element), making sure synchronization, allocation and scheduling are done at the right time, in the right order on each processing element. In short, each controller handles input from tasks, processes these and produce some output. The input is ready and finish signals. The output is run and preempt signals. The controller works by calling the synchronizer, allocator and scheduler in turn.

2.2.1 Synchronizer

The synchronizer handles dependencies between tasks. The direct synchronization protocol is used. This means that a task with multiple dependencies, will not be synchronized until all of its dependencies have been resolved. This is also known as AND dependencies.

The dependencies are modelled as a two dimensional structure of size N $\times$ N, where N is the total number of tasks on all processing elements. Each place of this structure: $dep_{i,j}$, model whether there is a dependency from task i to task j with true or false and dep_{ij} where $i = j$ is not defined. This structure is called *origdep* and remains the same once a system has been defined. In order to set and resolve dynamic dependencies, another structure of the same size called *dyndep* is used. The *dyndep* structure changes when task become ready and finish.

When the synchronizer is invoked by the controller, a possible finish signal, and a possible number of ready signals have been received by the operating system. The dependencies of all tasks that have become ready, are set copying the data from the origdep into the dyndep structure for the particular tasks. The dependencies of task j=α is set using the following operation:

$$dyndep_{i,j=\alpha} := origdep_{i,j=\alpha}, \quad i = 1 \dots N, \alpha \in [1 \dots N]$$

These dependencies are removed from the dyndep structure along with task finishing. When task i=β finishes, all tasks depending on it, will have their dependencies resolved, by setting them to false in the dyndep structure. This is done by the following operation:

$$dyndep_{i=\beta,j} := 0, \quad j = 1 \ldots N, \beta \in [1 \ldots N]$$

In this way a task $\tau_{j=\gamma}$ is said to be synchronized if

$$dyndep_{i,j=\gamma} = 0, \quad i = 1..N, \gamma \in [1 \ldots N]$$

Meaning that it has no unresolved dependencies.

2.2.2 Allocator

The allocator handles exclusive access to local resources on each processing element. When a task uses a resource, it becomes non-preemptive by other tasks using the same resource. Even though a task of higher priority becomes ready and synchronized, it will not run if it needs an all ready used resource. This is called Priority inversion and results in a schedule, where higher priority tasks are more likely to miss their deadlines. Several algorithms exist in order to give an upper bound guarantee of the Priority Inversion [11], in order to make high priority tasks finish in time more often. Our model implements three allocation algorithms:

Preemptive Critical Section This algorithm is basically not doing anything, as it only prevents tasks needing a used resource from being allocated. The priority inversions can be of arbitrary length, and the system may deadlock.

Non-Preemptive Critical Section If a running task uses a resource, it becomes non-preemptive, even if the higher priority task does not need its resource. The advantage of this protocol, is its simplicity, setting an upper bound on priority Inversion and preventing deadlock due to resource contention.

Priority Inheritance This algorithm allows higher priority tasks not contending for resources to preempt lower priority tasks even though they are using a resource. Like the preemptive critical section algorithm, deadlock is not prevented.

These allocation algorithms have been chosen in order to be able to display the capabilities of including an allocator in the MPSoC model.

In our allocation model, a task uses its resource in the entire execution time. This prevents transitive blocking [11] in the Priority Inheritance protocol and also prevents deadlock in the case of Preemptive Critical section and Priority Inheritance protocols. The issues of tasks using resources, in parts of their execution times only, is discussed in Chapter 6.

Timing anomalies may occur when tasks contend for a global resource. A precise definition of timing anomalies is given in [17]. The type of timing anomaly that may occur in our system is a Scheduling Timing Anomaly, where a local best-case execution time of a task, leads to a global worst-case schedule.

Even though the model only handles local resources, the communication on busses, resembles a global resource and a timing anomaly may occur, when different task contend for the bus. Our model is able to catch these timing anomalies and an example hereof is given in Section 5.1.17.

2.2.3 Scheduler

The scheduler uses priority-driven scheduling, which is a dynamic scheduling principle that evaluates the priorities of tasks in run time as they change. Other scheduling mechanisms are clock driven scheduling and Round Robin scheduling.
Because clock driven scheduling uses a static schedule calculated at compile time, the reactive nature of the MPSoCs, which are modelled, is not captured. Furthermore a new schedule must be devised every time the execution platform or application is changed.

Round Robin scheduling is a simple dynamic scheduling principle where each task is given a fixed amount of execution time each period. In weighted Round Robin scheduling, different execution times can be specified. The Round Robin principle does not take the dynamic urgency of tasks into account.
Priority-driven scheduling has been chosen for the above reasons. It is noted that our implementation in UPPAAL can fairly easy be extended to use the weighted Round Robin scheduling.

The scheduler chooses a task to run, among the allocated tasks, according to one of the following scheduling algorithms:

Fixed Priority (FP) Each task has a fixed priority.

Rate Monotonic (RM) The tasks are ordered after shortest period.

Deadline Monotonic (DM) The tasks are ordered after shortest static deadline.

Earliest Deadline First(EDF) The task with the earliest deadline, at the scheduling time has highest priority.

The first three scheduling algorithms are static, meaning that the priority of each task is known when the task is created on the platform. Using the EDF algorithm each task changes priority as time progresses, hence a dynamic scheduling algorithm.

2.2.4 Controller

The controller is seen in Figure 2.6. Firstly the processing element handles an input-part, where it receives a possible finish signal and a number of ready signals from the tasks mapped onto the *pe*. This is shown with a 1) in Figure 2.6. When all[2] signals have been received at all *pe's*, each of them make a synchronization in order to find all tasks which do not have unresolved dependencies. Then the allocator checks if all needed resources are available. Finally the scheduler finds the task with the highest priority. This is shown with a 2) in the figure. Finally run and preempt signals are send out to the right tasks, marked with a 3) in the figure.

2.2.4.1 Global correct schedule

Running the synchronizer, allocator and scheduler in turn on each processing element might not give a correct schedule. The following example shows this: Take a system $\{\tau_1\ ,\tau_2\} \mapsto\ pe_1$ and $\tau_3 \mapsto\ pe_2$ and the dependency relation $\tau_3\ \prec \tau_1$ where τ_1 has higher priority than τ_2. Due to the dependency, τ_2 is scheduled first on pe_1. In this example, it is assumed that τ_2 still has run time, when τ_3 finishes. From a global point of view τ_1 should be scheduled on pe_1, because it has higher priority than τ_2, marked a) in Figure 2.7. This will however not be the case, because os_1 on pe_1 is not activated (does not receive any ready or finish signals), marked b) in Figure 2.7.

[2] The next chapter describes how it is ensured that all signals are received.

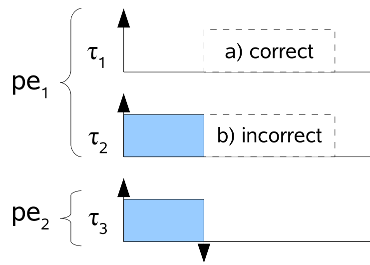

Figure 2.7: a) Global correct schedule. b) Global incorrect schedule.

Even though pe_1 has been activated by an input, the schedule could still be incorrect, if os_1 completes its scheduling before os_2, because os_2 has not resolved τ_1s dependecy.

In order to handle resolved dependencies on other processing elements, the concept of a reschedule is introduced. A reschedule means, that if a dependency has been resolved on a processing element, all other operating systems needs to do a new allocation and scheduling, taking tasks that have just been synchronized into account. The different approaches to do this reschedule is described below.

The simplest approach is to issue a reschedule signal from the operating system, on which a task that resolves dependencies finishes. This reschedule can be send to all operating systems which have tasks with resolved dependencies. This is done for each operating system that has a finishing task that resolves a dependency. In this approach the operating system may schedule one task τ_1 and send the appropriate run and preempt signals. After a reschedule the operating system may decide that another task τ_2 should run instead and preempt τ_1. Since each processing element can possibly resolve a dependency, M reschedules can occur in a system with M processing element. Since this is done before the next time unit starts, the schedule will be correct from a global point of view.

Even though the above strategy gives a correct schedule, it seems undesirable that tasks can be preempted in the same time unit as it is scheduled to run, and more than one run signal can be issued by an operating system in the same time unit[3]. In order to prevent this, we benefit from the processing elements being synchronous as stated in the top of Section 2.2. Instead of having the operating systems synchronize, allocate, schedule and then output run and preempt signals independently, the idea is to stop and wait at a barrier after the scheduling,

[3]This approach was used in the proposal by A. Brekling in [4]

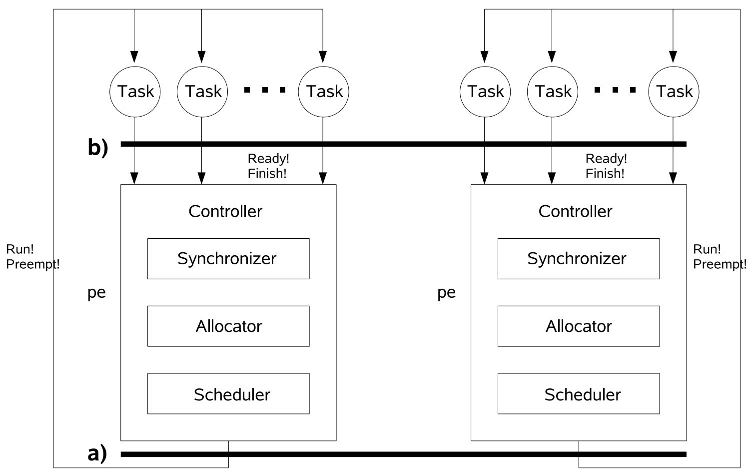

Figure 2.8: Two operating system with the two barriers a and b.

just before signals are send out. If a dependency has been resolved during the synchronization, a global reschedule flag is set. When all operating systems have met the barrier they will all reschedule if the flag is set. If the flag is not set, they will proceed by sending out run and preempt signals to the appropriate tasks. The synchronization barrier is marked **a)** in Figure 2.8.

2.2.4.2 Ready and resolved dependency in same time unit

Another issue to investigate, is a task coming ready in the same time unit as its dependency is resolved. As explained in Section 2.2.1, a dependency is set by the operating system of a task sending a ready signal. The dependency is resolved by the operating system of the finishing task. A special case arises when a task becomes ready in the same time unit as its dependency is resolved.

Look at the following example, where $\tau_1 \mapsto pe_1$ and $\tau_2 \mapsto pe_2$ and the dependency relation is $\tau_1 \prec \tau_2$. Both task have periods and deadlines on 4 and execution times of 2. The offset of τ_2 is 2 such that it becomes ready at time 2 when τ_1 finishes. If the operating system os_1 runs before os_2 the dependency is set by os_1 and resolved by os_2. τ_2 will become synchronized and run at time unit 2, as seen in Figure 2.9, where $\uparrow$ indicates the start of a tasks period and $\downarrow$ indicates a task finishing.

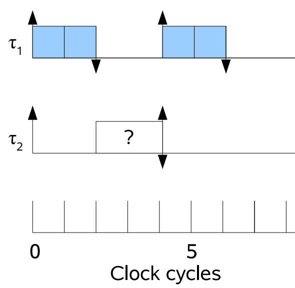

Figure 2.9: The possible scenarios when a task becomes ready in the same time unit as its dependency is resolved.

If, on the other hand, os_2 runs before os_1 the dependency is resolved first by os_2 (not actually doing anything) and then set by os_1. This results in τ_2 not being synchronized after two clock cycles, and therefore miss its deadline. The schedule is determined by which operating system runs first. This is not acceptable because we have chosen, that a system should be scheduled in one way only, when the execution times of tasks are fixed. It must be decided whether a task should become synchronized if it becomes ready in the same time unit as its dependency is resolved, and always do the same thing.

Since we have modelled the operating system to run in zero time, and that when a task finishes, it is truly finished and have possibly written some data, that another task depends on, it is made possible to be synchronized in the same time unit as a dependency is resolved. The special case where a task has its dependency resolved when it becomes ready, is therefore the best case for the task to actually meet its deadline in time. We will investigate what happens if the task is not synchronized until next time its dependency is resolved. The assumptions of this system are as follows:

1. The periods, p_1 and p_2 of τ_1 and τ_2 are the same and is equal to their deadlines.

2. The offset, o_1 of τ_2 is the finish time of τ_1. In this case we assume the offset of τ_1 to be zero and the finish time of τ_1 becomes its execution time e_1.

3. The dependency $\tau_1 \prec \tau_2$ becomes resolved the second time τ_1 finishes at time $p_1 + e_1$.

τ_2 is schedulable if its finish time, f_2 is equal to or earlier than its deadline d_2: $f_2 \leq d_2$. The deadline of τ_2 is its offset o_2 plus the deadline which is equal to its period p_2. The finish time, f_2 of τ_2 is equal to the start time of τ_2 plus the execution time, e_2 of τ_2. Using assumption 3, the start time of τ_2 becomes $p_1 + e_1$. We then have:

$$p_1 + e_1 + e_2 \leq o_2 + p_2$$

Using assumption 1 and 2, we get:

$$p_1 + e_1 + e_2 \leq e_1 + p_1$$

which is reduced to

$$e_2 \leq 0$$

τ_2 is only schedulable if its execution time is less than or equal to zero. As this is not possible, τ_2 is not schedulable even if its predecessor τ_1 finish at the most optimal time, already at time o_2. The above scenario advocates towards making task become synchronized when their dependencies are resolved in the same time unit as they become ready. This is ensured by setting all dependencies from ready tasks, before any dependency is resolved due to a finish signal. We model this using a barrier in the same way as the reschedule issue described above. The barrier used for this is marked b) in Figure 2.8.

2.2.5 Communication

Intra-processor communication among tasks is assumed to be specified implicit in the task, and is therefore modelled like a normal dependency. Inter-processor dependencies means that data calculated by a task on one processing element must be used by a task on another processing element. In this case the data is modelled to be transferred on a bus connecting the processing elements. In this MPSoC model a bus is modelled in the same way as a processing element and connects two or more processing elements. A communication on the bus is modelled as a special task in the following way. Consider a system where $\tau_x \mapsto pe_a$ and $\tau_y \mapsto pe_b$ with the dependency $\tau_x \prec \tau_y$. In this system there is an inter-processor dependency and the communication of data from τ_x to τ_y must be modelled on a bus. This is done by defining the communication

as a message task τ_m (introduced in [12]) running on a processing element p_m simulating the bus. The communication can start after τ_x has finished and τ_y must wait for the communication to finish. This is modelled by changing the dependencies to $\tau_x \prec \tau_m \prec \tau_y$. The bus is modelled, such that once a communication has started, it must finish before other communications can start. This is achieved by having all message tasks running on the bus using the same resource r_m which prevents preemption of any message task.

2.2.6 Processor frequency

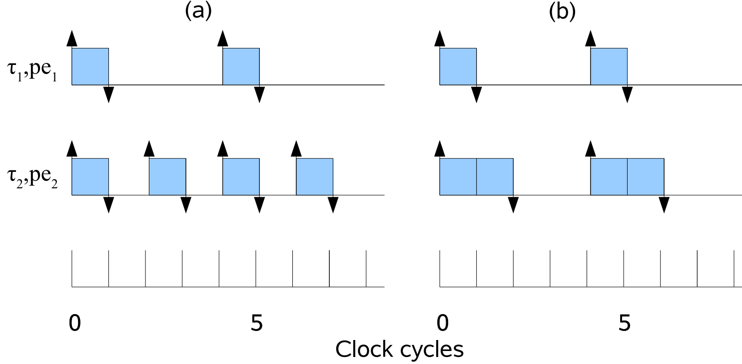

Figure 2.10: a) Periods are not comparable between processing elements. The period of τ_1 has become twice the period of τ_2. b) Periods are comparable.

The exact values for best and worst case execution time (*bcet* and *wcet*), given in execution cycles, are dependent on the architecture of the processing element. Once a task is mapped onto a processing element, the properties period (p), deadline (d) and offset (o) can be calculated in execution cycles by multiplying with the frequency of the processing element.

$$
\begin{aligned}
d_{i,j} &= f_i * \delta_j \\
o_{i,j} &= f_i * \omega_j \\
p_{i,j} &= f_i * \pi_j
\end{aligned}
$$

The execution times *bcet* and *wcet* of a task τ_j, running on pe_i is now directly comparable to $d_{i,j}$, $o_{i,j}$, $p_{i,j}$. These properties of the task is however not comparable to those tasks mapped on another processing element with a different speed, because each time unit represents different time intervals on each processing element. Take an example of τ_1 and τ_2, both with π of 2 and $bc\epsilon$ and

$wc\epsilon$ of 1. If τ_1 is mapped onto a processing element pe_1 with a frequency, $f_1 = 2$ and τ_2 is mapped onto a processing element pe_1 with frequency, $f_2 = 1$, the equations above give us that $p_{1,1} = 4$ and $p_{2,2} = 2$. Even though the periods were the same in seconds to start with, τ_2 is now running twice as often as τ_1 as seen in Figure 2.10a. This is not correct. In order to make $d_{i,j}$, $o_{i,j}$, $p_{i,j}$ comparable between processing elements, we will need to stretch the timing properties of some tasks, in order to make a time unit of the same size on all processing elements. Since we are only working with whole time units we need to use the *Least Common Multiple* (lcm) of the processors frequencies. The deadline, offset, period, best-case execution time and worst-case execution time are now calculated as follows:

$$
\begin{aligned}
d_{i,j} &= lcm * \delta_j \\
o_{i,j} &= lcm * \omega_j \\
p_{i,j} &= lcm * \pi_j \\
bcet_{i,j} &= \frac{lcm}{f_i}\ bc\epsilon_{i,j} \\
wcet_{i,j} &= \frac{lcm}{f_i}\ wc\epsilon_{i,j}
\end{aligned}
$$

Where lcm is the least common multiple of all processor frequencies. The result can be seen in Figure 2.10b.

This concludes our modelling of the MPSoC. We have described how the application consists of dependent, periodic and non-periodic tasks, with hard, firm and soft deadlines. The execution times of tasks are defined as a random value, between best and worst case. The operating system of each processing element (with possible different frequencies) are described, by the synchronization protocol, allocation protocol and scheduling algorithm. In the next chapter we define the exact semantics in UPPAAL which implements the behavior of the MPSoC as described in this chapter.

CHAPTER 3

Timed-Automata Semantics

In this chapter the formal semantics of the MPSoC model is presented. The structure of the UPPAAL semantics resembles the structure of the model described in the previous chapter. The UPPAAL model consist of a number of timed-automata templates, each representing a component of the MPSoC. As we will see, the interesting components are the Task template and the Controller template of the execution platform. The full semantics, including functions and declarations, can be found in Appendix F. This chapter assumes, that the reader has a general knowledge about timed automata. If not, an introduction is given in Appendix B. The formal model enables the possibility of verification using model checking.

Along with the formal model, the cost model, used for modelling memory usage and power consumption, will be described.

Next an explanation of our definition of a simple environment will be given.

In the end of this chapter, the special case where all tasks are assumed to execute in worst case execution time is described. A modified version of the model checker is used, giving a significant increase in the size of systems that can be feasibly verified. A special finish state is explained aswell, which is used for producing the schedule for schedulable systems as described later.

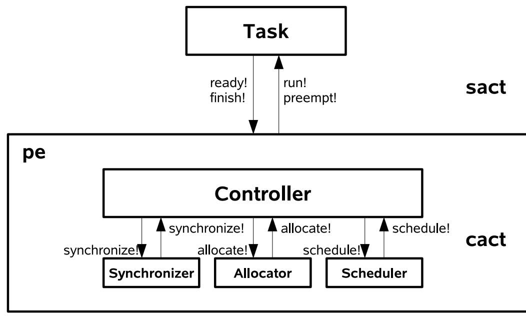

Figure 3.1: Channels used in UPPAAL

In this chapter all notions of execution cycles, deadline, offset and period refers to the properties e, d, o , p measured directly in execution cycles. Refer to Section 2.2.6 for the definition of these.

3.1 UPPAAL model

The complete MPSoC semantics in UPPAAL consists a number of parallel templates. The composition can formally be described as follows:

$$
\begin{aligned}
&System = Application \parallel ExecutionPlatform\\
&Application = \parallel_{j=1}^{n} \tau_j\\
&ExecutionPlatform = \parallel_{i=1}^{m} pe_i \parallel Rescheduler\\
&pe_i = Controller_i \parallel Synchronizer_i \parallel Allocator_i \parallel Scheduler_i
\end{aligned}
$$

where $\parallel$ means parallel composition of templates in UPPAAL. The only template, which does not reflect the modular structure of the MPSoC, is the Rescheduler template. This will be explained in connection with the Controller template.

Communication between a task and the Controller (operating system) is done though the tasks *sact* channel. As there are m processors, the *sact* channels are collected in a channel array of size m. Each channel is connected to its Controller. Each task on the processor uses this same channel. ready and finish signals flow from the task to the Controller, while run and preempt signals flow the other way. The *sact* channels are synchronous, because reaction to the signals must be taken immediately. Internally the controller activates its

synchronizer, allocator and scheduler through the *cact* channel. The channels can be seen in Figure 3.1.

The UPPAAL templates uses predicate functions on guards and assignments. A guard predicate function is evaluated to true or false. The predicates are named in an intuitive way, describing the conditions that must be satisfied for taking the transition. This gives a high level of abstraction, making the UPPAAL semantics more understandable at template level. The exact meaning of each predicate is found in Appendix F.

3.1.1 Application

Each task in the application is represented by one UPPAAL template[1]. The template has the same states as described in Figure 2.2 in the previous chapter. Before the template can be presented, the issue of how to model time must be addressed. Remember that each time unit, the task is looping in each of the states Idle, Ready and Running as seen in Figure 2.2.

3.1.1.1 Timing

Various approaches have been tried to model time of a task, *cp*, and making the task take a round each time unit in the states Idle, Ready and Running (as described in Figure 2.2). *cr*, the remaining execution time, need to be an integer variable since it is not possible to either stop a clock, or save the value, in the case of preemption.

1. The first approach is to include a local clock x, which together with the use of invariants and guards, forces the task to take one round each time unit. This clock only needs to run in the interval [0;1]. The time passed since the beginning of a tasks period *cp*, is also modelled as a clock and is reset when it becomes equal to the tasks period (when a new period starts). This approach uses two clocks for each task template.

2. A fairly simple improvement of the first approach, is to change the clock *cp* into a variable, and manually increase it by one, on the transitions where time progresses. The clock x, still runs in the interval [0;1], except in the IdleWait and Idle states as described later. Reducing the number of clocks from two to one per task, has given a significant improvement in

[1] In the proposal by A. Brekling, the task consisted of four templates

the size of systems which can be feasible verified, as clocks are one of the main sources for complexity in timed automata.

3. Having taken the first step towards removing clocks, the question arises if the use of the local clock x is really necessary. The clock is used for two things: To make sure that tasks move round each time unit (decrementing the dynamic scheduling criteria and *cr* if the task is in the running state) and that they all do it at the same time. In this way, the local clock x has the same function as a double barrier construction described in [1], preventing race conditions. The third approach uses a central unit, sending signals to all tasks, making them take a round each "time unit". There is really no time unit, but the behavior becomes the same as if there were.

Removing the second clock should also increase the size of feasible systems. However in the model with local clocks, there were no need to make an actual loop in the state Idle. Time will pass by, and the task will issue a ready signal when *cp* or x becomes equal to the period. The model checker uses a simplification, collapsing time when a task is in the same state as time passes. The improvement of removing the second clock seems to be cancelled by the time collapse. The second approach has been used, because it preserved the decentralized design of the MPSoC model, where tasks only communicate with their own processing elements.

3.1.1.2 The Task Template

Because of the complexity in the Task template. The description is split into three parts. First the states that directly resemble the model in Figure 2.2 are presented. Also the initialization and the state used for a task with an offset is explained here. This makes up the basic task model. Next it is described how the state for missed deadlines is modelled. Finally the task model used for non-periodic tasks is explained.

Basic task template
A task moves around in the three main states, Idle, Ready and Running as described earlier in Section 2.1. If the task has an offset, it moves to the Offset state, where it waits until the clock x becomes one less than the offset. Then the clock x is set to the tasks period minus one moving to the IdleWait state. The task finally moves to the Idle state, at the latest, just before x becomes equal to the period. This is ensured by the guard $x > ptime - 1$ on the transition from IdleWait to Idle, and the invariant $x < ptime$ in the IdleWait state. The reason for the extra state IdleWait is shortly explained. In the rest of the description, combinations of guards and invariants that are used to make the task change

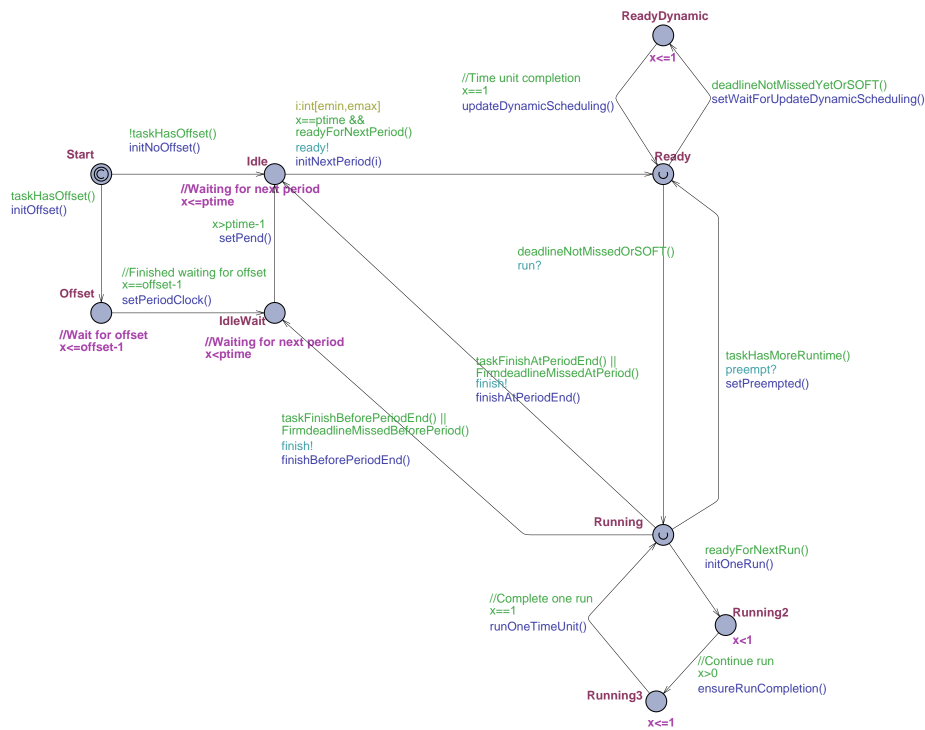

Figure 3.2: The basic states and transitions of the task template

state at specific time are omitted, as they work in the same way as just described.

The operating system must be able to receive all ready and a possible finish signal when it is running. Therefore all tasks sets a lock called *Pending* just before the ready signal is send to the operating system. This is the reason for the IdleWait state. The statement *setPend()* in the transition from IdleWait to Idle sets this lock, making sure that its operating system receives the signal at the start of the next time unit. Another solution for implementing this, could be to omit the IdleWait state and make a loop in the Idle state, with a conditional statement setting the *Pending* lock exactly one round before the next period. However in creating the extra state IdleWait, a loop transition is omitted each time unit for all Idle tasks. This is also the reason for the two transitions from Running to Idlewait and Idle, when a task is finished (cr is equal to zero). In the special case where a task finish at the end of its period, the *Pending* lock is set and the task moves directly to the Idle state. If the task finished before the end of its period, it moves to the IdleWait state, until one time unit before the

start of a new period, as described above.

Sending and receiving signals (moving between Idle, Ready and Running) only happens in the beginning of a time unit. Because the operating system runs in zero time (only committed and urgent states) this ensures that all ready and finish signals are send to Controller, then all operating systems are run, and finally run and preempt signals is send from controller, before any task takes a new round in Ready and Running.

The non-deterministic execution times of tasks, are modelled in the following way. The transition where a task issues a ready signal and sets cr to its execution time, is divided into a number of non-deterministic transitions, each setting cr to a different value in the interval $[bcet;wcet]$. This is done using the Select statement in UPPAAL.

Detection of missed deadlines

First hard deadlines are described. In the proposal by A. Brekling, deadline detection is done by making the system deadlock, putting the invariant $cp < deadline - cr$ on the Ready state. In this way the system will deadlock if there is not enough time to finish the task.

We have chosen another approach. First of all, the detection of deadlines is done by moving the task to the MissedDeadline state. Missed deadlines are detected by querying if any task will ever move into this state. Not querying for deadlocks, gives the possibility of assigning priorities to each UPPAAL template, which will be explained later. As the system does not deadlock, the features of soft and firm deadlines also becomes possible. Instead of making a task miss its deadline when it becomes impossible to finish in time, we wait until the actual deadline is missed and $cp \leq deadline$ becomes false. By moving the task into the MissedDeadline state, at the exact time of the deadline miss, the schedule produced by the MoVES tool will also be correct.

The MissedDeadline state can be seen to the far right side, of Figure 3.3. Even though it becomes impossible to finish in time, already when $cp > deadline - cr$, the task do not miss until $cp > deadline$. In this way, the task may also be in the running state when it misses a deadline. The transitions from Ready and Running to the MissedDeadline state, has the predicate `deadlineMissed()` in the figure.

Tasks with firm deadlines have the possibility of missing a deadline either in the Ready state or in the Running state. In either case the task should move back to the Idle or IdleWait state, in order to save execution time on the processor. Finished tasks, move to the IdleWait state if they finish before the end of their period, and move to the Idle state if they finish at the end of their period.

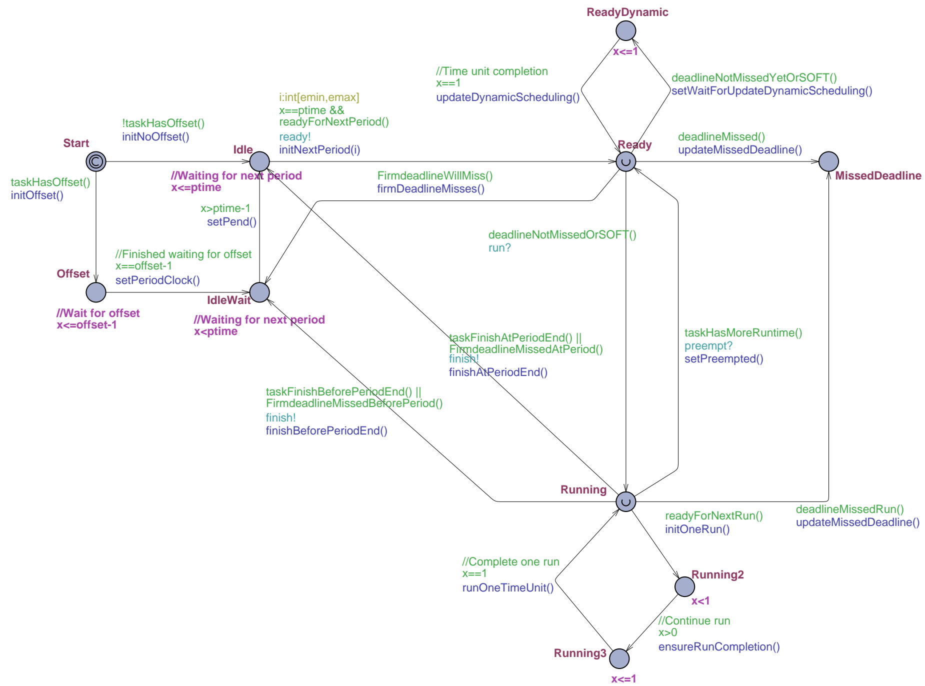

Figure 3.3: The task template including deadline miss

In the same way, a task with firm deadline in the Running state, which misses before its period, moves to IdleWait. If it misses at the period, it moves directly to the Idle state, setting the *Pending* lock in the same way as finishing tasks with hard deadlines. In both cases, a finish signal is send to the Controller, in order to free up the processor. This finish signal will however, not resolve dependencies, as the firm task did not actually finish.

The same should be the case from the Ready state. However in order to save a transition in the template, we detect a miss from the Ready state, exactly one time unit before it actually misses. It is therefore only necessary to move the task to the IdleWait state.

Soft deadlines are straight forward, as nothing special happens to the task. The task stays in its state as the soft deadline is missed. In this way, the dynamic criteria is still updated after the deadline is missed and hereby becomming negative. In the case of Earliest Deadline First scheduling, this means that the time of its deadline is before the current time, which makes sense.

Non-periodic tasks
The non-periodic tasks have the same structure in the Ready and running states as the periodic tasks. The template is seen in Figure 3.4. When a non-periodic task finishes it goes back to the start state and waits for an input on its triggering channel. The Pending lock is set, ensuring that the Ready signal, is received and handled, by its operating system.

A triggered task can be triggered by either the Controller or an Environment. The triggered task supports the same kinds of deadlines as the basic Task.

3.1.2 Execution Platform

In the execution platform , the Controller part is most interesting. The Allocator, Synchronizer and Scheduler are activated from the controller in turn, by sending the *synchronize*, *allocate* and *schedule* signals. These units perform their operations and return control to the controller by sending the same signals back. This can be seen in the middle part of Figure 3.5. The operations of synchronizing, allocating and scheduling becomes simple operations in this way. The modular structure however makes it fairly simple to for example, change the way scheduling is done, by replacing a single function in the scheduling template. The only interface are the channels *synchronize*, *allocate* and *schedule*. The templates of the Synchronizer, Allocator and Scheduler are presented and described briefly.

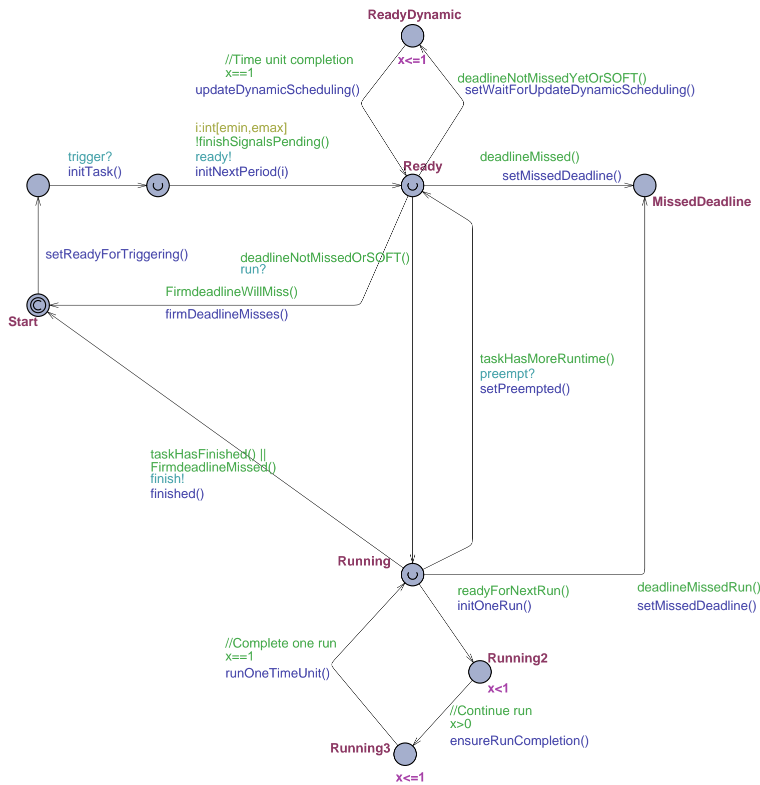

Figure 3.4: Non-periodic task template.

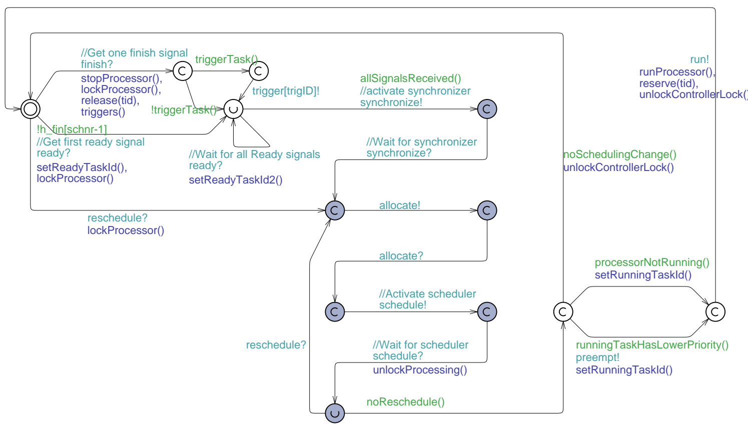

Figure 3.5: The Controller template.

3.1.2.1 Controller

The Controller template represents the operating system and can be divided into three parts. This can be seen in Figure 3.5. In the first collection of white states, input signals (ready and finish) are received. In the middle part, with gray states, the actual operating system is run, synchronizing, allocating and scheduling. In the last part, output signals (run and preempt) are handled.

Because the operating system runs in zero time, all states in the Controller, are either urgent or committed. This means that time is not allowed to pass, while a template is in such a state. The start state is an exception.

As described in the previous chapter, barriers are needed in order to synchronize the controllers for two reasons. The possibility of a reschedule, and making sure all dependencies are set before any dependency is resolved. The barriers are implemented as the two urgent states seen in Figure 3.5. All controllers wait in the first white urgent state, until all input signals have been received (making sure all dependencies are set). In turn all controllers go through the synchronization, allocation and scheduling and waits in the gray urgent state. If one or more dependencies have been resolved during the synchronization, all controllers are forced to reallocate and reschedule. The reschedule signal is send from the special Rescheduler template. This template fires reschedule signals on a broadcast channel to all controllers, when they have all entered the urgent state. After the reschedule, all controllers take turns to send out possible run and preempt signals and return to the initial state waiting for input signals.

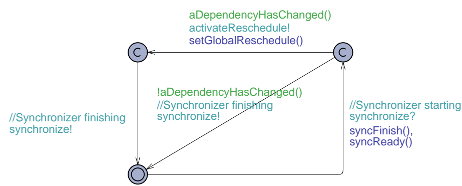

Figure 3.6: The Synchronizer template

Below the internal processor part is described. The templates differ from the model proposed in [4], because sequential code has been collected in functions, instead of using states and transitions to do calculations. This gives a better overview of the templates, and makes them easier to extend with more scheduling and allocation algorithms.

3.1.2.2 Synchronizer

The Synchronizer template is seen in Figure 3.6. It is activated by the Controller, and starts by synchronizing finished tasks, and hereafter tasks which has become ready. The Synchronizer stores information about tasks, which have resolved their dependencies. If a task has no unresolved dependencies, it will be synchronized.

If a dependency has been resolved, the Rescheduler will be activated. By activating the Rescheduler it is ensured, that the task with highest priority and no dependencies will be scheduled on each processor.

Finally the Synchronizer will finish, and send a signal back to the Controller.

3.1.2.3 Allocator

After the Synchronizer, the Allocator, seen in Figure 3.7, will be activated. The allocator only handles tasks, which have been synchronized. If a task needs an available resource, it will become allocated. If the resource is used, the task will not be allocated.

If the Priority Inheritance protocol is used, a low-priority task using a resource, will inherit the priority of a task with higher priority, if the task needs the

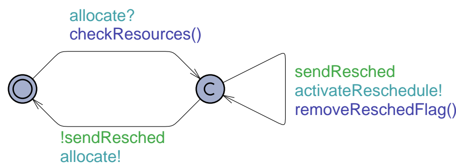

Figure 3.7: The Allocator template

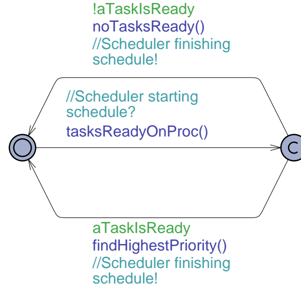

Figure 3.8: The Scheduler template

same resource. Hence it could be necessary to reschedule after allocating (if the task using the resource has been preempted by another task which does not use the resource). After allocating, the allocated tasks is stored in the array: `allocated`, and the Allocator can activate the Rescheduler if there have been any priorities inherited.

3.1.2.4 Scheduler

After having synchronized and allocated, the Controller activates the Scheduler. The scheduler chooses the task with the highest priority according to the desired scheduling algorithm. The supported scheduling algorithms can easily be extended by changing the function: `findHighestPriority()`.

The Scheduler will only perform a scheduling, if there are allocated tasks on the processor. In either case it will pass control back to the Controller by sending a signal to it. As seen in Figure 3.8, the Scheduler can finish in two ways, either choosing a task to run, or if no tasks have been allocated.

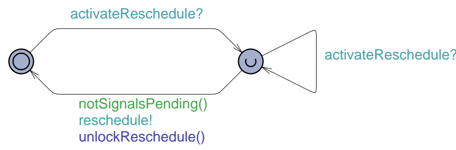

Figure 3.9: The Rescheduler template

3.1.2.5 Rescheduler

In order to handle rescheduling, the concept of a Rescheduler template has been introduced. The Rescheduler template is global and handles rescheduling for all processors. The Rescheduler is able to catch one or more rescheduling signals as seen in Figure 3.9. When all controllers have finished their processing phase (synchronization, allocation and scheduling), the Rescheduler sends the rescheduling signal.

The Rescheduler sends a broadcast signal to all processors at once, making them do a reallocation and rescheduling right away.

3.2 Environment

The environment interacts with the MPSoC through the channels of the non-periodic tasks. In this section a simple environment is described, which can interact with a task. A separate instance of the environment template is created, for each non-periodic task, interacting with the surroundings. The environment template is pretty simple as seen in Figure 3.10, which is useful when generating the environment through the MoVES tool described later.

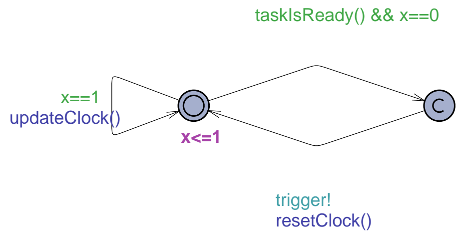

Figure 3.10: The Environment template.

The environment template created, triggers its non-periodic task, at any time after the minimum inter-arrival time. If the minimum inter-arrival time is set to 1, the triggering can happen at total random times.

3.3 Cost

As resources such as Memory and Power are very limited in embedded systems, it is relevant to model the use of these resources in our MPSoC design. The model keeps track of the cost in each time unit. In this way, it can be verified, using model checking, that a cost never exceeds a certain limit. For example that the total power consumption of all processing elements does not exceed a certain limit. The schedulability of an MPSoC can also be considered a cost which can be verified to be within the schedulable limit.

A task has a certain cost (memory and power), depending on which state it is in. The tasks adds their cost to the total cost on its processor, at the beginning of a time unit and subtracts it at the end of the time unit. Because all tasks move around together each time unit, there is one global state each round, that holds the total cost on the processor.

An example: A running task uses some memory for its calculations. When it finishes, and has written whatever data needed to its local memory, the task goes into the Idle (or IdleWait state) and does not take up the memory anymore.

The total cost on each processor can be found by adding up the costs for all tasks, which are mapped onto it. Adding up these processor costs of the whole MPSoC is also useful, as the total power consumption is a relevant feature to verify.

The cost model is constructed, so the user of the model can specify his own costs, by adding a cost to each state of all tasks in the application. The definition of cost_i for task_j becomes:

$$Cost_i(\tau_j) = (Start_{ij}, Idle_{ij}, Ready_{ij}, Running_{ij}, Preempted_{ij})$$

Using this state based generic cost model, memory usage and power consumption is modelled in the following way, according to the definitions of the MPSoC model.

The static memory, *sm*, is the program code and is added once and for all

in the Start state of the task. When the task runs, the dynamic memory, *dm*, is used for calculations performed by the task. If the task becomes preempted, all data is stored, in order for the task to resume once it becomes running again. The dynamic memory *dm*, is therefore still used when the task is preempted. There is no actual preempt state in our model, but a flag is set when the task moves to the Ready state from the Running state at a preempt signal. In this way, the cost model also models the state of preemption. In our MPSoC model, no dynamic memory is used, when the task is Ready or Idle (also IdleWait).

A very simple approach for modelling power has been taken. When a task is running, it uses power, *pw*. In all other states the power consumption is zero. A more realistic approach for modelling power usage is proposed in the future work chapter. The memory and power costs are modelled in the following way for each task:

$$Cost_i(\tau_j) = (Start_{ij}, Idle_{ij}, Ready_{ij}, Running_{ij}, Preempted_{ij})$$
$$Memory(\tau_j) = (sm_j, 0, 0, dm_j, dm_j)$$
$$Power(\tau_j) = (0, 0, 0, 0, pw_j)$$

In addition to this model, a special feature for the memory cost is captured. After a task finishes, having produced some data, there is no way to guarantee that the data is read, by a task on the same processor or a message task on the bus, straight away. Look at the following example. Three tasks with descending priorities, τ_1, τ_2, τ_3 and the dependency relation $\tau_1 \prec \tau_3$ and τ_3 must use some data produced by τ_1. All tasks run on the same processing element. After τ_1 finishes, τ_2 is scheduled. During the running time of τ_2, the data produced by τ_1, must be saved until it is read by τ_3 and is modelled to be included in τ_3. This is also included in the cost model, and is called shared cost.

Even though the cost model has been constructed with memory usage and power consumption in mind, it is generic and can be used for other models of memory usage and power consumption. Also other costs can be defined. This concludes the cost model. The syntax for defining cost can be found in Appendix C.

3.4 Model checking

This section is divided into two parts. The main focus is verification that all tasks meet their deadlines and this is described first. Secondly the notion of response of a task is explained and how it can be verified to be within a certain upper bound.

In order to detect missed hard deadlines in an MPSoC system in UPPAAL, model checking is done in the following way. For all possible executions of the UPPAAL model, does the model eventually go into a state, where a task is in the MissedDeadline state? $\exists \diamond$ *missedDeadline*, where the predicate *missedDeadline* is a boolean function that becomes true, if one or more tasks have moved into the MissedDeadline state. In UPPAAL this is expressed as `E<>missedDeadline`. If the answer is *Property is not satisfied*, no deadlines are missed and the system is schedulable. If the answer is *Property is satisfied*, some task has eventually moved into the MissedDeadline state. In this case it is possible to get a trace, which shows an example where the property is satisfied.

Some of the non-determinism in the model can be removed, thereby reducing the complexity of the model checking. The non-deterministic choices in the model can be divided into two categories:

1. When a task becomes ready and the execution time is set to a random value in the interval $[bcet;wcet]$.
2. The non-deterministic choices of which template to choose first when:
   - The operating systems execute their receive phase (ready and finish signals)
   - The operating systems execute their send phase (run and preempt signals)
   - The tasks taking a round in the Ready and Running state.

The second type of non-deterministic choices, where the model checker chooses a task or operating system to move one state, results in the same schedule by definition of our model. The schedulability of the system is only dependent on the properties of the MPSoC system and not on the order of these non-deterministic choices of the second type. We exploit this, by assigning unique priorities to each template, only enabling one transition at a time. In this way, the computation tree of the model checker, becomes less complex.

In the special case where all execution times are worst case, further optimization becomes possible. In this case, there is only one possible schedule for the system. This results in only one trace. If the system misses a deadline, a task will eventually move to the MissedDeadline state. In order to detect this, the model checker actually do not need to save all visited states. This is not the case when the system is schedulable. Here the model checker, continues until an already visited state is encountered again and the schedule loops. It is then

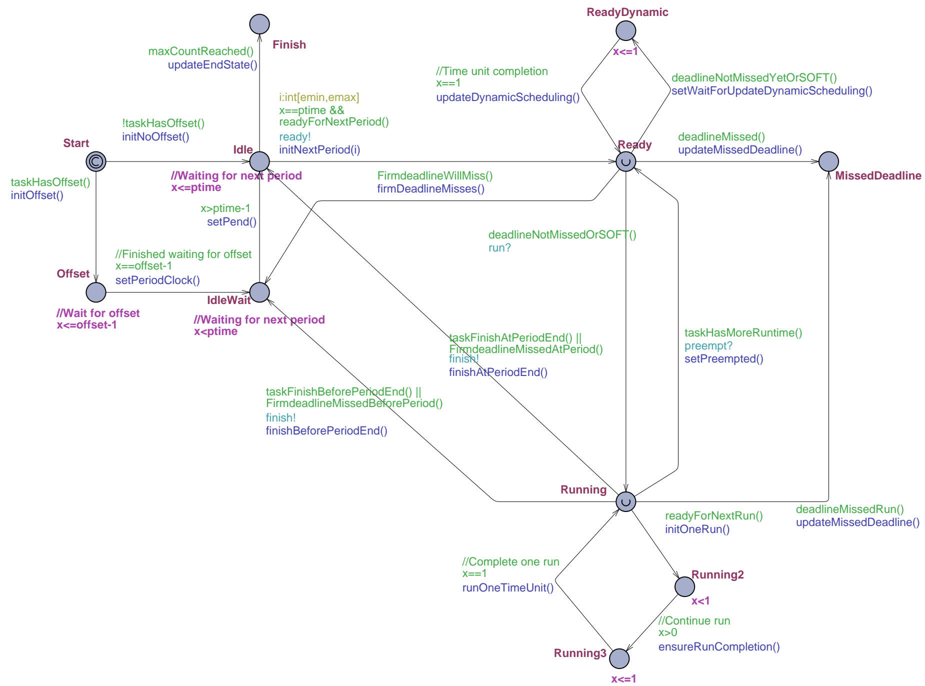

Figure 3.11: Task template including the Finish state.

certain, that no tasks can ever go into the MissedDeadline state.

Even though there is only one trace through the system, the problem of enormous memory use, arises when the periods of systems are very long, since all states are still saved. If we can find the maximum time at which the schedule is going to repeat itself, we can make the model go into a special Finish state at this time, identifying schedulability, without exhaustive search of the state space. The query used to detect this would be `E<>allFinish()`. Using this state it is also possible to dump the trace when the system is schedulable because an example trace can be given when an exists-property ($\exists$) is satisfied. This trace is used to produce an understandable version of the schedule. In fact, it is no longer needed to store all visited states, as we are sure the model will eventually either go into the MissedDeadline state, or the Finish state. The Finish state can be seen in Figure 3.11

It is intuitive correct that the schedule will repeat itself after some time. According to [2], in a single processor system, with preemptive tasks, with offset and a deadline which is not necessarily equal to the period, the schedule repeats at the latest after time:

$$2H + max(\pi)max(\delta)$$

where H is the least common multiple of all the periods. This is based upon the fact that the system is predictable with a fixed operating system creating the same schedule, when the parameters of the task are the same. In our case we have multiple processing elemnts with inter-processor dependencies, and local resource allocation. The schedule of this system will however also repeat after the same amount of time, since everything is static. This is not formally proven but is supported by Doctor Colin Perkins at University of Glasgow.

A modified model checker for UPPAAL where visited states are not saved, has been developed by J. Illum from Aalborg Universitet. This enables verification of large systems with very long periods.

3.4.1 Response time

The response time of a task is defined as the time between the task becomes ready and the time it finishes. The response time is in this way dependent on the scheduling of the task on the processing element. It is relevant to verify the maximum response times of tasks with soft deadlines. This can be done by querying that the variable *cp* of task x will never be greater than a certain limit y. In a query this is written `E<>TaskcX.cp<Y`.
This concludes our description of the timed-automata model in UPPAAL. The next chapter describes oyr strategy for evaluating this model.

Chapter 4

MoVES tool

Having defined the UPPAAL semantics (refer to Chapter 3), we are able to create the tool MoVES (Modelling and Verification of Embedded Systems), which uses the model. MoVES, as stated in the objective, is meant to assist in designing, verifying and reconfiguring MPSoCs in an intelligent way. The user needs to have an understanding of the MPSoCmodel, but not necessarily of timed automata. It is assumed that tasks and their timing properties etc, is already defined and therefore MoVES is only concerned with helping the designer reconfiguring the execution platform and the mapping of tasks on it.

The first section in this chapter describes the flow in MoVES. How the user interacts with MoVES, how MoVES creates the UPPAAL-model and which results MoVES gives the user.
The second section will describe the structure of MoVES, and how the different classes interacts. The complete source code can be found in Appendix E or on the enclosed CD-ROM. A HTML-API for the MoVES tool is enclosed on the CD-ROM.

4.1 Flow in MoVES

MoVES is constructed as a frontend to Verifyta, which is a commandline-based model-checker for UPPAAL-models. The flow through MoVES is illustrated in Figure 4.1. In the following section each step through the tool will be described.

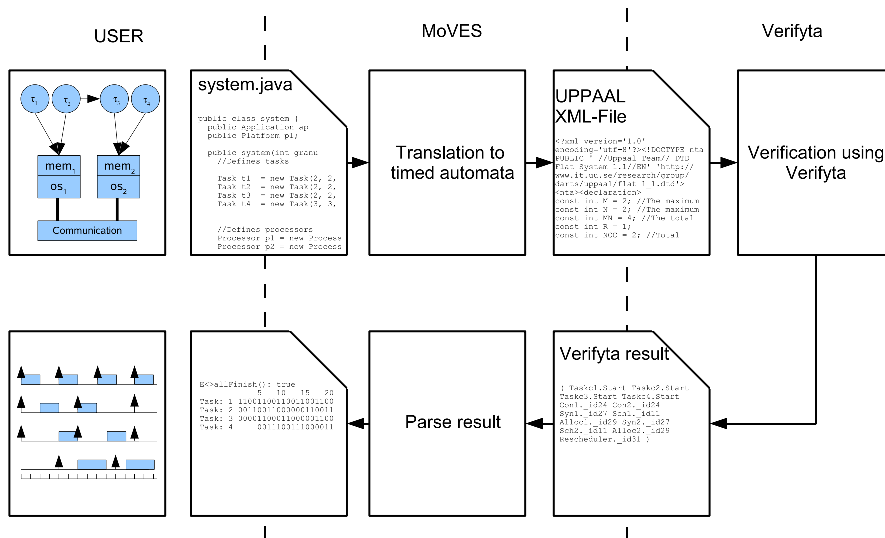

Figure 4.1: Illustration of flow in MoVES

4.1.1 Deriving code from taskgraph

Knowing the application, platform and mapping of tasks onto the platform, the system can be defined in MoVES. This is done in the file `MPSoC.java`. All tasks must be specified with $bc\epsilon$, $wc\epsilon$, δ, ω, π and fp (For further description on defining systems in `MPSoC.java` see Appendix C).

When the system is defined, MoVES is run using Java Runtime Environment. MoVES gives different results (see Section 4.1.5), according to the input arguments (for details about arguments, refer to: C.2.2).

4.1.2 Generating the UPPAAL XML-system

When an MPSoC-system is defined in `MPSoC.java` and MoVES is run, the UPPAAL-model used in the verification is created. First periods, offsets and deadlines in cycles for each task is calculated. This is done according to the formulas written in Section 2.2.6, if the user has specified a granularity for the system, this is included in the calculation. After having calculated periods, deadlines and offsets, the total utilisation is calculated on each processor.

4.1.2.1 Granularity

The user can give a granularity as argument for MoVES, if the number of executioncycles etc. is to high[1]. All periods, offsets, deadlines and execution times for the MPSoC will be divided by the granularity and rounded up. In this way, larger systems become feasible to verify.

This may have an influence to the schedulability in the following way. If a system with high granularity is schedulable, the corresponding system with a lower granularity is schedulable. This is not true the other way around. Because the execution times are rounded up when dividing by the granularity, the total utilization of the system is also increasing to a possible non-schedulable level. Further timing anomalies may dissapear becase executiontimes is divided by the granularity and rounded up as well. This may result in *bcet= wcet*.

4.1.2.2 Utilisation

In MoVES there is a build in check, making sure the total utilisation of hard-deadlined tasks on each processor is below 1.00[2]. The check verifies if the system is non-schedulable without using Verifyta.

$$\sum_{i=1}^{n} \frac{task_i.e}{task_i.p}$$

By using the above equation, the utilisation is calculated on each processor. n is the number of tasks on each processor. If the utilisation on one of the processors, then the system is not schedulable. This is written to the user, and the system is not written to the XML-file and verified using Verifyta. If the utilisation of hard-deadlined tasks is below 1.00, the XML-file is created and verified.

[1]UPPAAL only supports 16 bit integers [-32768 ; 32767]

[2]If utilisation above 1.00, the system is not schedulabe because this means the executiontime on the processor is higher than the period.

If wanted, the user can have the utilisation for each processor displayed.

4.1.3 Verification in Verifyta

Now having created the XML-file, Verifyta is used to verify the system. Verifyta uses the XML-file and a file containing queries[3] as argument to the verification. The file can contain one or more queries. Among with this file, a couple of other parameters for Verifyta is used:

-S 2 used to optimize space consumption (level set to: most)

-T is used to activate reuse of statespace when several queries is verified.

-t 0 activates trace, 0 means "some trace" which will be the first reached. Only used when schedule is activated in MoVES.

-y Displays a symbolic post-stable trace, which can be read into MoVES. Only used when schedule is activated in MoVES.

The first two arguments is used to reduce memory use, hereby increasing the size of verifiable systems. While the last two arguments is used when a schedule from the verification is desired.

The result from the verification is piped into a TXT-file, which is read from MoVES after Verifyta has finished. If the users wants a schedule, Verifyta writes an extended trace[4] onto `stderr`, which is read and parsed by MoVES.

4.1.4 Parse result

Having a TXT-file containing the result of the verification, MoVES finds the results for each query. This is done by first finding all queries and hereafter searching the TXT-file for the results for each query. With the arguments for Verifyta specified earlier, only the following results can be given:

- Property is satisfied

[3]The queries is written in Computational Tree Logic (CTL)

[4]Containing both positions and variables in each state.

- Property is NOT satisfied
- Out of memory

These are the textstrings, which is searched for in the TXT-file. When one of these strings is found, the result is stored along with the corresponding query.

If the schedule option is activated, MoVES uses the `Parser class`. From `stderr` all lines are read and send to the parser. In the parser, all tasks position are found. Once each timeunit, the position of all tasks is stored. It is noted,

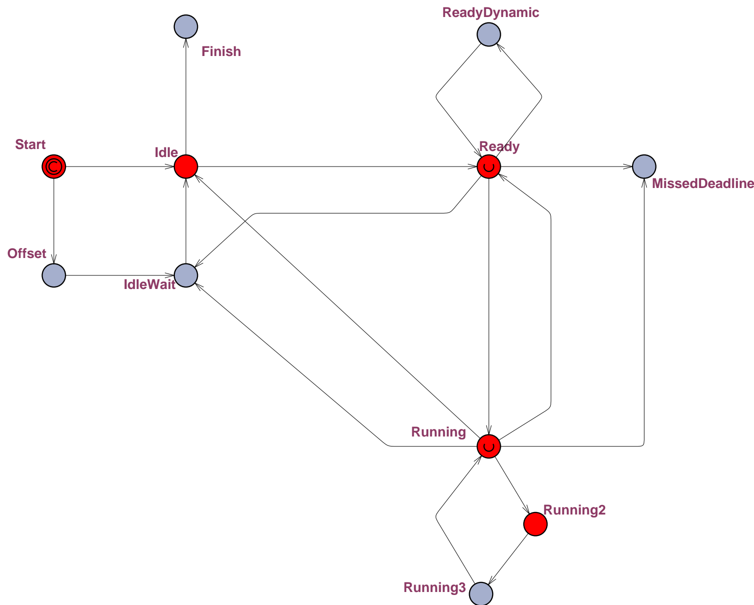

Figure 4.2: Shows states which is used when parsing the schedule

that if no tasks is positioned in any of the red states in Figure 4.2, the next transition taken will lead to the beginning of a new timeunit.
The state used in the schedule is the last, before a task enters a new timeunit. In order to find one unique state each timeunit, from which the position of the tasks can be stored, the red states of Figure 4.2 is used. If any task is positioned in one of the red states, this state is trown away. Most of the times, where all tasks is positioned in gray states is quite trivial to store in the schedule. These are the cases where atleast one task is located in either `readyDynamic` or `Running3`.

If this is not the case, and all tasks wait for the next period to start or off-

set to exceed, they are positioned in `IdleWait` or `Offset`. As described earlier (in Section 3.1.1) the clocks in Verifyta is collapsed and hence there is only one transition out of this state (when a task has finished its offset or should begin a new period), but it is not known for how long, the tasks are positioned in these stated. MoVES uses its own clocks to calculate how many timeunits to elapse in this state. When having the number of timeunits, all tasks are written as `idle` in the schedule.

4.1.5 Output

After Verifyta has finished the verification and MoVES has parsed the results from Verifyta, it is presented to the user. According to the arguments given to MoVES this result will change. Below an output, where 4 queries is verified, a schedule is wanted and the utilisation is printed along with an explanation of the schedule. MoVES is run using: `java MOVES -t -u -e`:

```
Utilization:
Processor 1:    0,67
Processor 2:    0,83
Verification time: 0.02 min
E<>missedDeadline: true
E<>totalCostUsed(Memory)>30: false
E<>totalCostUsed(Memory)>10: true
E<>totalCost[0][Power]>15: false
            5   10
Task: 1 11001100110
Task: 2 00110011000
Task: 3 00001100110
Task: 4 ----001100X

1 = running
0 = idle
- = offset
x = Missed deadline
X = Missed deadline running
* = finished
```

At the top of the output, the utilisation is written. As seen, no processors has a utilisation over 1.00.
Then the time used by Verifyta is written. In this case it took two seconds.
Below the results of the queries are written. As seen, a task misses a deadline,

the total memory used is in the interval between 10 and 30 units, and the use of power on the first processor (array index 0) does not exceed 15 units.
Next the schedule is printed. The schedule uses the notation written in the bottom. As seen Task 4 misses a deadline after 11 timeunits.

The output from MoVES can then manually be transformed into a graphical schedules, as seen in the bottom left corner of Figure 4.1.

4.2 Structure of MoVES

MoVES is written in Java. The class structure is of the MPSoC-class is similar to MPSoC-systems. The MPSoC class consists of a platform and an application. The platform consists of a number of processors, resources and environments, while the application consists of a number of tasks and costs. See the classdiagram in Figure 4.3.

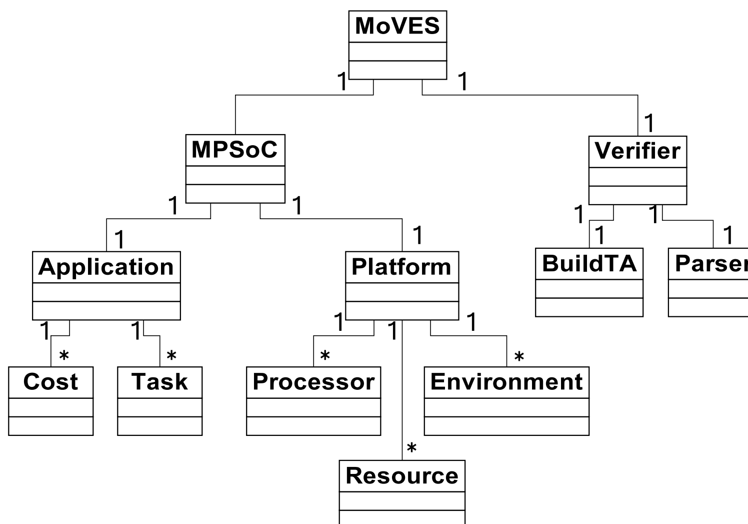

Figure 4.3: Class diagram for MoVES

The MPSoC class is instantiated by the MoVES-class. The instance of MPSoC is given as argument to the Verifier-class, from where an instance of BuildTA (Build Timed Automata) and Parser is created. Both BuildTA and Parser uses

the MPSoC object. The instace of BuildTA generates the XML-file containing the UPPAAL-model, the Verifier-class runs Verifyta, while the Parser object is used to parse the trace provided by Verifyta.

4.2.1 MoVES-class

The MoVES-class contains the actual main-class of the tool. All arguments for MoVES is handled here, and provided to the Verifier. It is also in the main-class the instance of the MPSoC is created. According to the given arguments MoVES will execute the according function in the Verifier class. The MoVES-class only handles inputs from the user. If the arguments are faulty, this is presented to the user, while results are handled by the Verfier.

4.2.2 MPSoC-class

An MPSoC is defined and generated in the MPSoC-class. The user specifies an MPSoC, which should be verified in the file: `MPSoC.java`. The class creates a number of different objects from the other classes: Task, Processor, Cost, Resources and Environment. These objects are mapped together in either the Application or Platform class.

4.2.3 Verifier -class

After beeing activated by the MoVES-class, the Verifier starts by creating the XML-file in the BuildTA class. First deadlines, offsets and periods are calculated according to the processor-speed. Secondly the XML-file is created.
Thridly the Verifier uses Verifyta to verify the queries provided in the query-file. Verifyta generates a file with results of the verification, and maybe a trace. The handling of trace is then done in the Parser-class, from which all lines on `stderr` is read.

Chapter 5

Evaluation

The evaluation of the system is divided into four parts. The first part is a test which should convince that the implementation is done according to the model described in Chapter 2. This is done by identifying a number of simple cases which together covers all the different features of the system. All these test cases use processing elements running at unrealistic slow frequencies such as 1 Hz and 2 Hz. This makes it feasible to manually create the expected schedule of the system and compare it with the output from the MoVES tool. All cases of this test performed as expected and the implementation in UPPAAL is therefore assumed to be done correctly according to the described MPSoC model.

All tests are done using the MoVES tool directly. In this way the MPSoC model and the implementation of the tool is tested at the same time.

The second part of this chapter describes a case study, which shows the application of the MoVES tool on a real life example. This example is a smart phone with five different applications (115 tasks all together) running in parallel on multiple processing elements.

In the third part, the maximum size of systems, which can be checked in feasible time and memory use, is investigated.

A comparison to the model proposed by A. Brekling in [4], has also been

made in terms of correctness and performance. These cases are known from literature or tested using TIMES tool [9].

The final section of this chapter contains a summary of the results.

All tests are carried out on a 3,4 GHz, with 2 GB of memory, running Windows XP. The computer uses java 1.5 and the version of Verifyta enclosed with UPPAAL 4.0.3.

5.1 Testcases

Below is given a short description of each testcase. The output of each case can be found in Appendix D. If nothing else is stated, all processors runs with frequencies of 1 Hz.

5.1.1 Highest priority first

Having two tasks running on same processor, the task with the highest priority should be scheduled first. This test has been done using all 4 scheduling algorithms.

Task	*bcet*	*wcet*	*d*	*o*	*p*	*FP*	Scheduled first
τ_1	2	2	6	0	8	1	RM, FP
τ_2	3	3	5	0	8	2	DM, EDF

5.1.2 Task finishes at end of period

A task finishes at end of its period, and therefore it should become ready at the same time. The tasks execution time, deadline and period is the same, hence the task executes all the time.

Task	*bcet*	*wcet*	*d*	*o*	*p*	*FP*
τ_1	4	4	4	0	4	1

5.1.3 Task with highest priority has dependency

The task with the highest priority will not be able to run before its dependency has been resolved. This test has been conducted on both single- and multi-processor systems. τ_1 depends on τ_2. Because of the dependency, τ_1 is not scheduled to run before τ_2 has finished, even though it has highest priority (using FP-scheduling). This is seen in Figure 5.1a.

Task	*bcet*	*wcet*	*d*	*o*	*p*	*FP*
τ_1	2	2	6	0	6	1
τ_2	2	2	6	0	6	2

5.1.4 Task depends on task with offset

A task τ_1 becomes ready, but depends on τ_2 which has an offset. This offset must elapse and τ_2 must finish before τ_1 can be scheduled. Other than the offset of τ_2, the systems is equivalent to the system in the previous case. The resulting schedule is illustrated in Figure 5.1b.

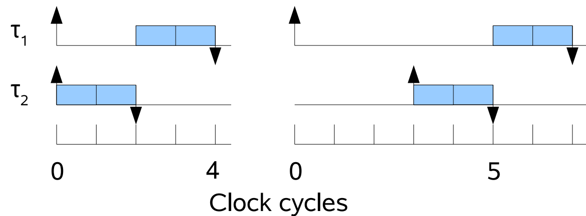

Figure 5.1: Task depends on lower priorities a) and b)

5.1.5 Ready and finish at same time

In this case, τ_2 depends on τ_1. Furthermore τ_2 has an offset which makes it becomes ready when τ_1 finishes. Both on one and two processing elements, τ_2 should be scheduled directly when it becomes ready (recall Section 2.2.4.2).

Task	*bcet*	*wcet*	*d*	*o*	*p*	*FP*
τ_1	2	2	4	0	4	1
τ_2	2	2	4	2	4	2

5.1.6 Chain of dependencies to resolve

In this case, there is a chain of dependencies, $\tau_1 \prec \tau_2 \prec \tau_3$. Meaning that τ_1 preceeds τ_2 and τ_2 preceds τ_3 (as seen in Figure 5.2). Even though τ_3 has the highest priority (using FP-scheduling), it will not be scheduled before τ_2 has finished.

Task	*bcet*	*wcet*	*d*	*o*	*p*	*FP*
τ_1	2	2	6	0	6	3
τ_2	2	2	6	0	6	2
τ_3	2	2	6	0	6	1

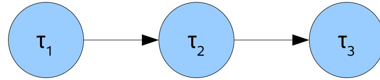

Figure 5.2: Chain of dependencies

5.1.7 Four tasks depends on one task

Four tasks depends on one task (τ_1). This means that the four tasks all should be able to run when τ_1 has finished. This test consist of two cases. In the first case, the tasks are mapped on the same processing element. In the second case, the tasks are mapped onto five different processing elements. In the second case, all processing elements will be idle from time unit 5 until time unit 10 each period.

Task	*bcet*	*wcet*	*d*	*o*	*p*	*FP*
τ_1	2	2	10	0	10	1
τ_2	2	2	10	0	10	2
τ_3	2	2	10	0	10	3
τ_4	2	2	10	0	10	4
τ_5	2	2	10	0	10	5

5.1.8 One task depends on four other tasks

One task depends on four other tasks. These tasks has no dependencies, and can be scheduled right away. The dependent task may not run before the four other tasks has finished. This test is also split into two cases. In the first case, the tasks are mapped on the same processing element. In the second case, tasks are mapped onto five different processing elements. In the second case all processing elements will be idle from time unit 5 until time unit 10 each period.

Task	*bcet*	*wcet*	*d*	*o*	*p*	*FP*
τ_1	2	2	10	0	10	1
τ_2	2	2	10	0	10	2
τ_3	2	2	10	0	10	3
τ_4	2	2	10	0	10	4
τ_5	2	2	10	0	10	5

5.1.9 Utilisation above 1.00

Having a utilisation above 1.00, at one processing element, will make the system not schedulable. The utilisation in these cases are 1.00 and 2.00. The first case can be scheduled, the other can not.

Task	*bcet*	*wcet*	*d*	*o*	*p*	*FP*
$\tau_{1,1}$	3	3	6	0	6	1
$\tau_{1,2}$	3	3	6	0	6	2
$\tau_{2,1}$	6	6	6	0	6	1
$\tau_{2,2}$	6	6	6	0	6	2

5.1.10 Several preemptions of the same task

One task (τ_5) is preempted several times by other tasks. Each preemption lasts two time units, and then τ_5 can run for two time units. The case is modelled with five tasks running on one processing element. As seen in Figure 5.3 the bottom task is preempted several times.

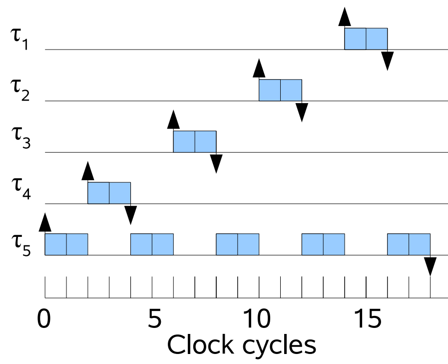

Figure 5.3: A task is preempted several times

5.1.11 Several preemptions of different tasks

Several tasks are preempted. Five tasks are running one the same processing element, and every two time units, a new task with higher priority is released. This will give a pyramidal execution as seen in Figure 5.4.

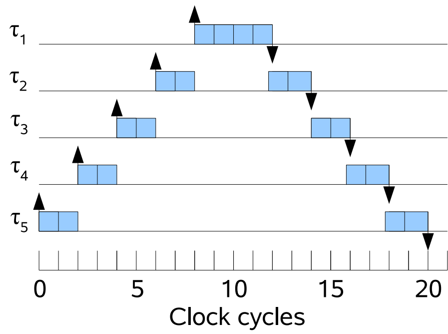

Figure 5.4: Several tasks are preempted

5.1.12 Finish leads to preemption

In this test there are three tasks on two processing elements, with the mapping $\tau_1 \mapsto pe_1, \{\tau_1, \tau_2\} \mapsto pe_2$ and the inter-processor dependency $\tau_1 \prec \tau_2$. When

τ_1 finishes, τ_2 will preempt τ_3, which has lowest priority. This can be seen in Figure 5.5.

Task	*bcet*	*wcet*	*d*	*o*	*p*	*FP*
τ_1	2	2	6	0	6	1
τ_2	2	2	6	0	6	2
τ_3	4	4	6	0	6	3

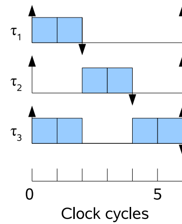

Figure 5.5: A task finishes leads to preemption

5.1.13 Shared resource, highest priority first

In this test, there are three tasks on one processing element. τ_1 and τ_3 uses the same resource r_1. τ_2 does not use the resource but has higher priority than τ_3. τ_1 with highest priority executes first. This scenario tests that the scheduler also works correct with resources.

5.1.14 Resource sharing protocols

The following cases, will test the allocation protocols. The following three sub-cases uses the same task set on one processing element. The tasks are scheduled according to the fixed priority scheduling.

Task	*bcet*	*wcet*	*d*	*o*	*p*	*FP*	Resources
τ_1	3	3	10	2	10	1	r1, r2
τ_2	3	3	10	1	10	2	
τ_3	3	3	10	0	10	3	r1

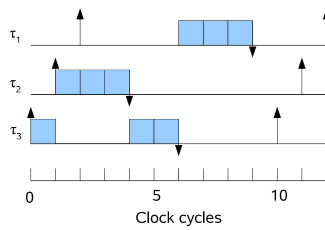

Figure 5.6: Schedule of Preemptive Critical Section

5.1.14.1 Preemptive Critical Section

τ_2 is able to preempt τ_3, because it has higher priority and does not use the resource. When τ_2 has finished, τ_3 will finish before τ_1 (with highest priority) resumes execution. This can be seen in Figure 5.6.

5.1.14.2 Non-Preemptive Critical Section

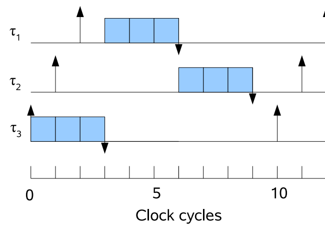

Figure 5.7: Schedule of Non Preemptive Critical Section

τ_3 starts executing. Due to the Non-Preemptive Critical Section τ_3 will run until it finishes. Hereafter τ_1 is scheduled because it has the highest priority. Finally τ_2 is scheduled. This is illustrated in Figure 5.7.

5.1.14.3 Priority inheritance

τ_3 will begin executing. After one execution unit, τ_2 will be ready. Since it has higher priority than τ_3, it will be scheduled and τ_3 preempted. After another time units τ_1 will be ready. Since it has the highest priority, and uses the same

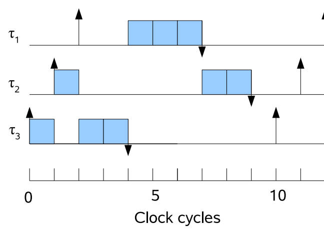

Figure 5.8: Schedule of Priority Inheritance

resource as τ_3, is is blocked and τ_3 will inherit τ_1's priority. Therefore τ_2 is preempted. When τ_3 has finished, τ_1 will execute, and finally τ_2 will execute the rest of its execution time. This can be seen in Figure 5.8.

5.1.15 Different processor speeds

In this case, 2 identical tasks is mapped onto 2 processing elements. The first processing element runs at 1 Hz while the other runs at 2 Hz. As described in Section 2.2.6 the properties of each task must be comparable. In this case, the example from 2.2.6 is used, and a schedule as in Figure 2.10b is expected.

5.1.16 Granularity

A test concerning the granularity has been performed. In this case, three tasks are mapped onto three processing elements:

Task	*bcet*	*wcet*	*d*	*o*	*p*
τ_1	25	25	100	0	100
τ_2	50	50	100	0	100
τ_3	75	75	100	25	100

With a granularity of 25, a result like Figure 5.9 is expected.

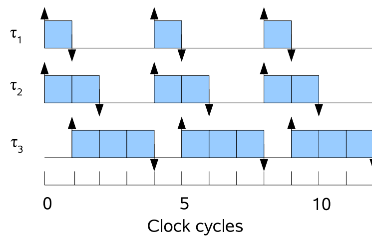

Figure 5.9: Expected result with a granularity of 25

5.1.17 Timing anomaly

Including resource usage, tasks can become non-preemptive and timing anomalies can occur [17] (when executing a task faster, leads to a missed deadline).

The case is carried out on two processing elements with two inter-processor dependencies. These dependencies is resolved using message-tasks. All processing elements runs with Fixed Priority scheduling. The tasks are defined below:

Task	*bcet*	*wcet*	*d*	*o*	*p*	pe	*fp*
τ_1	3	5	12	0	12	1	1
τ_4	4	4	10	0	12	1	2
τ_2	4	4	12	0	12	2	1
τ_3	3	3	12	0	12	2	2
τ_{m1}	2	2	12	0	12	bus	1
τ_{m2}	2	2	12	0	12	bus	2

The dependencies in the system are: $\tau_1 \prec \tau_{m2} \prec \tau_3$ and $\tau_2 \prec \tau_{m1} \prec \tau_4$.

If τ_1 executes in *bcet*, the system will miss a deadline. If τ_1 executes in *wcet*, all deadlines are met. This is illustrated in Figure 5.10 where *circ* indicates the deadline of a task.

5.1.18 Granularity prevents timing anomaly

In this case a granularity is set, which will prevent the timing anomaly described in the previous case. When setting the granularity parameter to two, all tasks

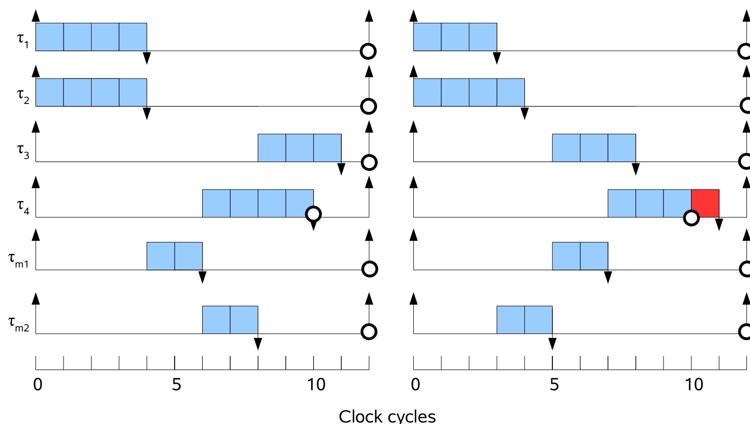

Figure 5.10: Timing anomaly, faster execution of τ_1, makes τ_4 miss its deadline

meet their deadlines. In this way, the granularity has made it impossible to detect the timing anomaly. The parameters for each task, after the granularity is set, is found in the table below. The schedule of the system after adding the granularity is seen in Figure 5.11.

Task	*bcet*	*wcet*	*d*	*o*	*p*	pe	*fp*
τ_1	2	3	6	0	6	1	1
τ_4	2	2	5	0	6	1	2
τ_2	2	2	6	0	6	2	1
τ_3	2	2	6	0	6	2	2
τ_{m1}	1	1	6	0	6	bus	1
τ_{m2}	1	1	6	0	6	bus	2

5.1.19 Soft deadline

In this case a task with a soft deadline misses by two time units each period. Eventhougt the task preceds τ_3 and misses its deadline, all hard deadlines are met. τ_1 and τ_2 is mapped onto pe_1 while τ_3 is mapped to pe_2. As described in Section 2.1.2, the periods of soft deadlined tasks may be skewed, and hence it is not possible to verify `allFinish()` properties if Soft deadlines is used. This means that it is not possible to create a schedule, but it would look like Figure 5.12.

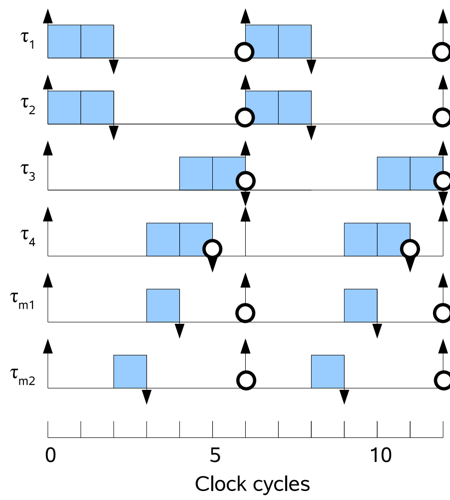

Figure 5.11: Granularity prevents timing anomalies

Task	*bcet*	*wcet*	*d*	*o*	*p*	Deadline type
τ_1	1	1	2	0	2	Hard
τ_2	3	3	4	0	4	Soft
τ_3	1	1	7	0	7	Hard

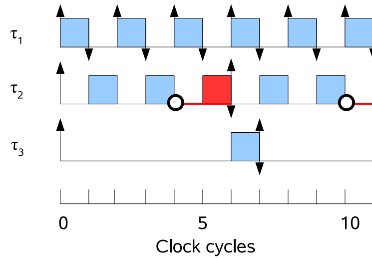

Figure 5.12: Schedule where Soft deadline misses, but system is still schedulable

5.1.20 Firm deadline

The firm deadlines is testet in four different cases. In the first case the task with firm deadline will be able to execute for 1 time unit (a and c), while in the

second case (b and d), the task with firm deadline will not be able to run. Both cases have been tested both with $d = p$ and $d < p$. This is done to verify, that the firm deadlined task is idle until the beginning of its next period, instead of becomming ready at once.

Task	*bcet*	*wcet*	*d*	*o*	*p*	Deadline type
τ_{a1}	5	5	6	0	6	Hard
τ_{a2}	3	3	6	0	6	Firm
τ_{b1}	6	6	6	0	6	Hard
τ_{b2}	3	3	6	0	6	Firm
τ_{c1}	5	5	8	0	8	Hard
τ_{c2}	3	3	6	0	8	Firm
τ_{d1}	6	6	8	0	8	Hard
τ_{d2}	3	3	6	0	8	Firm

5.1.21 Cost model

In the following cases, the cost model has been tested. Each state of the cost model has been tested seperately. Three tasks are used to test the model. Two subcases are tested for each case. Either the task runs for its entire period ($\tau_{1,1}$) or just a part of it ($\tau_{1,2}$).

Task	*bcet*	*wcet*	*d*	*o*	*p*	*fp*
$\tau_{1,1}$	3	3	6	0	6	2
$\tau_{1,2}$	6	6	6	0	6	2
τ_2	2	2	6	2	6	1

Each state of the model is verified one by one. 5 memory units will be used by τ_1 in the state currently verified. τ_2 is used when verifying ready, preempted and shared cost.

In each case, the cost of the different states is verified, along with information about the tasks position (e.g. is the task in `ReadyDynamic` when evaluating ready-cost?). The following queries will be used:

```
1. E<>Taskc1.IdleWait
2. E<>totalCostUsed(Memory)<5 && Taskc1.IdleWait
3. E<>Taskc1.ReadyDynamic
4. E<>totalCostUsed(Memory)<5 && Taskc1.ReadyDynamic
5. E<>Taskc1.Running3
6. E<>totalCostUsed(Memory)<5 && Taskc1.Running3
```

E.g. when verifying idle cost, and query 1 returns true, query 2 must return false (because the memory usage in `idleWait` is 5 units). If the task finishes at end of its period, it will not be located in `idleWait` and hence both queries will return false.

5.1.21.1 Static cost

In the static model, the task should add its static cost when it is instantiated. Here after it should not be removed again.
This is tested, by asking if τ_1 in states `IdleWait`, `ReadyDynamic` and `Running3` is below 5. If not, the static cost is added as expected.

5.1.21.2 Idle cost

Secondly the idle cost is tested. This is done by verifying if the cost in the `idleWait` state is below 5 units. This time it is also verified, if the task is in the `idleWait` state at any time. In the first subcase, both properties is expected to be true, while in the second subcase, both is expected to be false, since the task will finish at end of its period and never be idle.

5.1.21.3 Ready cost

The ready cost is tested in the same way. Again by verifying if the task is in the `readyDynamic` state, the cost should not be below 5 units. In the first subcase, it is expected that the task is in `readyDynamic` and the cost is not below 5 units, while in the second subcase, the queries should be false because the task is scheduled directly when it becomes ready.

5.1.21.4 Running cost

This time, the task should have a cost when it is running. Since no tasks can be modelled runnning 0 units, the cases that τ_1 is in state `running3` should be validated to true in both subcases.

5.1.21.5 Preempted cost

The preempted cost is modelled, so the cost should be verified in the readyDynamic state. In the first subcase τ_1 starts running, until τ_2 exceeds its offset. τ_2 preempts τ_1 and the cost is verified. In the second subcase, τ_2 is not added to the system, and therefore no preemptions occur.

5.1.21.6 Shared cost

The shared cost, is modelled as cost written by τ_1 to τ_2. Now τ_2 is modelled with a 5 units offset, which will make τ_1 finish two units before τ_2 becomes ready. In these two units the cost should be saved.

5.1.22 Environment

In this case a simple environment is tested. The environment determines when the non-periodic task is triggered. The environment is modelled to trigger the non-periodic task at any time after its minimum arrival time[1]. The minimal interarrival time in this example is set to 1, meaning the environment can trigger at any time.

Including the environment will drastically increase the verification time. As seen in Appendix D the verification times, when testing either `allFinish()` or `missedDeadline` are much greater when an environment is included (increase from 1 second to nearly 3 minutes).

The verification takes longer time due to an increase in complexity of model checking, because the environment can trigger the non-periodic task nondeterministically at any time after its minimum inter arrival time.

5.2 Smart phone

This section describes how MoVES can be used on 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, JPEG-encoder, JPEG-decoder and MP3-decoder (114

[1]The triggering may happen between the minimum inter-arrival time up till infinity.

tasks). 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 in the smart phone example, origin from experiments done by M. Schmitz [19]. The timing properties, period and deadline, of the tasks, are imposed by the application. The voice of the smart phone must for example be encoded and send 50 times each second, giving the GSM-encoder a period and deadline of 0.02 seconds. The execution cycles, memory usage and power consumption of each task is architecture dependent. These properties of the tasks, have been measured, by simulating the execution of each task on different kinds of processing elements.

A task graph for the MP3 decoder [10], along with a sketch of the original description in C, can be seen in Figure 5.13 [19]. In this example the decoder must perform all its operations and output a frame of real stereo audio each 25 ms. Within these 25 ms, several transformations such as *Huffman Decoding* are applied to the encoded input. Each transformation is modelled as a task. The flow of data between tasks can be seen as directed edges. As opposed to the C specification, the task graph exhibits parallelism and seems to fit on two processing elements. It is noted that in Figure 5.13a two columns of transformations exists. One for each channel of sound (stereo).
The data used in this example originates from measurements done by M. Schmitz.

The five applications have been mapped onto 6 processing elements connected by a bus. The parallelism of the MP3-decoder has been used to split this application onto two processing elements. The rest of the applications run on their own processing element. It is possible that there exists smarter designs, where less processing elements are used or one or several processing elements run at lower frequency. This is however not the focus of neither this case study nor the thesis in general. We have however shown that our tool is capable of verifying systems of realistic size. This means both the number of tasks and processing elements, and also the timing properties of the various applications are of realistic size. The verification of the smart phone example took roughly 2.5 hours.

The source code for generating the smart phone in MoVES can be found in Appendix E.13. The next section tests the size limit of the tool.

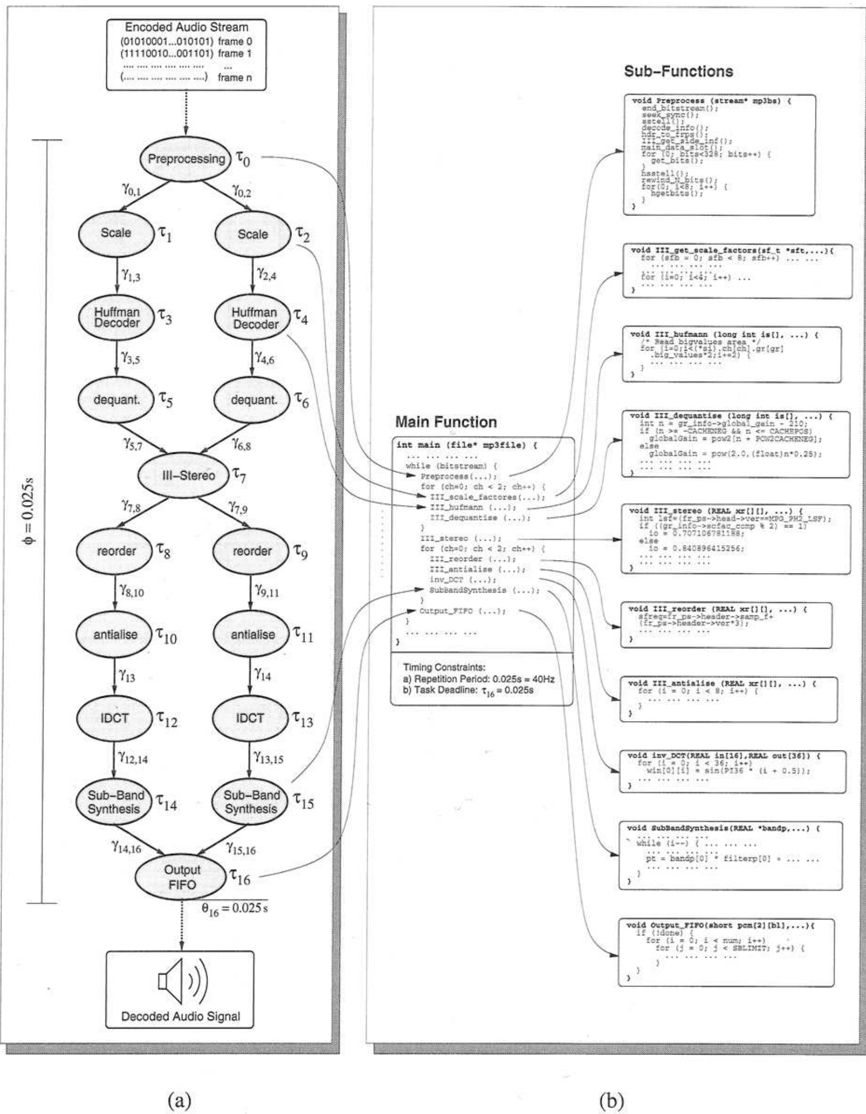

Figure 5.13: Task graph and C code sketch for MP3 decoder taken from [19].

5.3 Feasible Sizes

In this section, the feasible sizes of systems which can be verified is tested. The first test is done on a nondeterministic system with $bcet \neq wcet$. Secondly the deterministic model is tested ($bcet = wcet$) with the original version of Verifyta. Thirdly, the model including the finish state is tested. Finally the version of Verifyta provided by Jacob Illum is tested. The tests with the special Verifyta have been carried out on a corresponding machine, in order to compare the results with the rest. The machine for this is a 64bit Linux server with dual core AMD processor and 2 GB ram[2].

5.3.1 Non-deterministic model

First a test of the non-deterministic model has been carried out. Through this test it was shown that two factors played an important role in the complexity and thereby the size of systems which can feasibly be verified.

- The number of nondeterministic choices.
- The length of the periods for the verification

In the first case a number of identical tasks with period and deadline of 200 and [bcet;wcet] = [1;2] were tested. Figure 5.15a shows the verification time growing exponentially as a function of the number of tasks very fast. Figure 5.14 shows the maximum number of tasks that can be verified before the computer goes out of memory.

In the second case the number of tasks is constantly 2, and the size of the interval [bcet;wcet] is variable. Figure 5.15b shows the verification time as a function of the size of the interval.

5.3.2 Worst-case execution with Original Verifyta

The test of the original version of Verifyta has been divided into two parts. In the first part the finish-state of the task-model is not used. In this case Verifyta decides when the verification is done[3]. In the second part of the test, the finish-state is used. In this case, the worst case number of execution cycles (static

[2] The Linux clusters on IMM-DTU

[3] Either if the property is satisfied, or if a state is met, were Verifyta has already been.

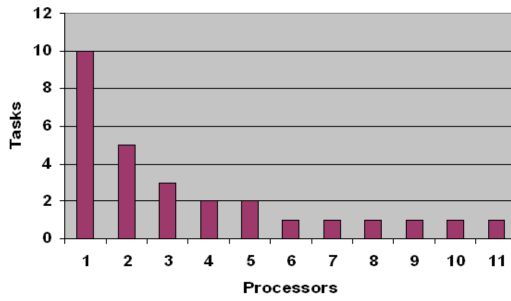

Figure 5.14: Verifiable nondeterministic tasks on processing elements

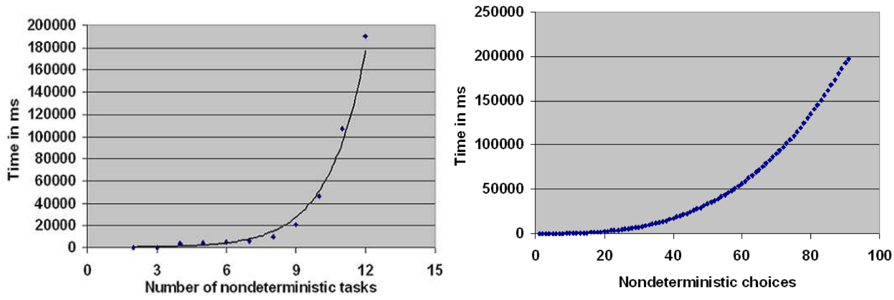

Figure 5.15: a) Verification time as function of nondeterministic tasks b) Verification time as number of nondeterministic choices

stop time), in which a task must miss its deadline if it ever will, is static as argued in Section 3.4. If no task miss its deadline before this time, the system is considered schedulable. Figure 5.16 shows the maximum number of tasks, which can be verified, as a function of processing elements. Using the model checker seems to be slightly more efficient. This is most likely due to the static stop criteria, which is based on an absolute worst case. Calculating the static stop time makes use of a ceiling function making some tasks run longer than necessary. The period and deadline of all tasks are 200. And the execution times $bcet = wcet$is 2. The maximal number of tasks verified was 176 with finish state and 196 without the finish state.

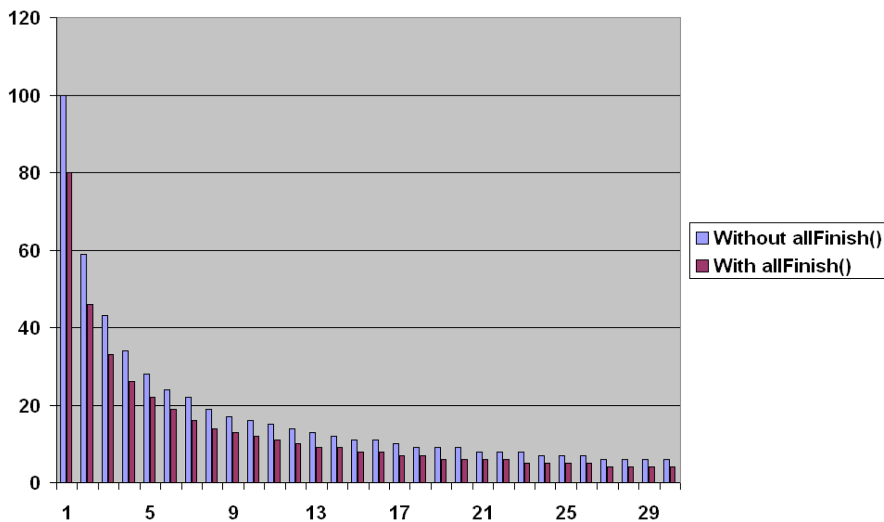

Figure 5.16: Diagram of verifiable systems with and without end state

5.3.3 Worst-case execution with special Verifyta

Finally a test of the special Verifyta provided by Jacob Illum. All tasks in this test also has a period and deadline of 200 and an execution time of 2. Since the special Verifyta does not store any states through the verification, the model must use the finish-state with the static stop criteria. Since the special version of Verifyta only runs on linux and our java implementation is developed for Win XP, only a few examples of feasible sizes and verification times has been obtained. These can be seen in Figure 5.18. 400 tasks can for example be verified in about 10 hours. As seen in Figure 5.17 the use of memory is linear. An estimate of the number of verifiable tasks would according to this about 1800 tasks. But as seen in Figure 5.18, time for the verification is growing exponentially. 600 tasks is verified in about 65 hours.

5.4 Comparison

Finally a comparison, to the model proposed by A. Brekling in [4], has been carried out. The first test cases will examine the scheduling algorithms and the functionality of the model. The last case, Multiprocessor example 3, contained

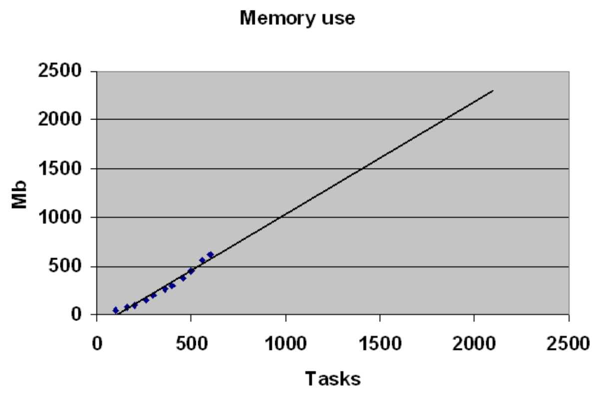

Figure 5.17: Diagram of verifiable systems with special Verifyta and an estimate of verifiable systems

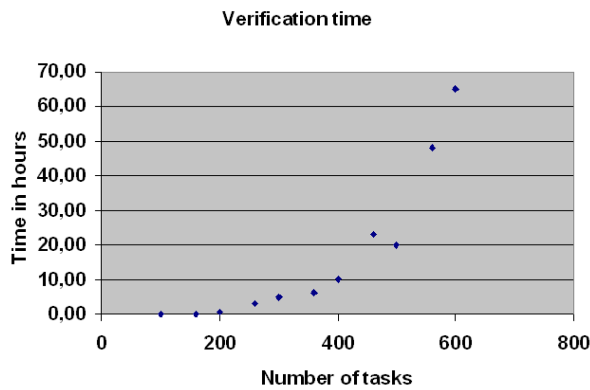

Figure 5.18: Diagram of time used on verifying systems with special Verifyta

the maximum number of tasks that could be verified in 2 minutes. In comparison our model handles this example in less than one second.

5.4.1 Singleprocessor example 1

This case consists of 3 tasks on a single processor system. The system can be schedulable using either Rate Monotonic or Earliest Deadline First scheduling.

The definitions of the tasks are found below:

Task	*bcet*	*wcet*	*d*	*o*	*p*
τ_1	6	6	10	0	10
τ_2	6	6	20	0	20
τ_3	2	2	30	0	30

5.4.2 Singleprocessor example 2

This example is similar to example 1, but now the execution-time for τ_3 is increased by one. This will make the system not schedulable using Rate Monotonic scheduling, but schedulable using Earliest Deadline First.

5.4.3 Singleprocessor example 3

Again same example as above, now with another increase of the execution-time for τ_3, to 4 units. This will make the system not schedulable, no matter which scheduling algorithm is used. As seen in the appendix, the result is: *Utilisation above 1.00, system not schedulable.*

5.4.4 Singleprocessor example 4

In this case it is illustrated how Deadline Monotonic scheduling can be used to make a system schedulable. The system is schedulable using Deadline Monotonic scheduling, but not Rate Monotonic. The properties for the tasks is found below:

Task	*bcet*	*wcet*	*d*	*o*	*p*
τ_1	3	3	5	0	5
τ_2	2	2	4	0	6

5.4.5 Singleprocessor example 5

In this case a intra-processor dependency is examined. The example is schedulable using Earliest Deadline First, but not using Rate Monotonic scheduling. There is a dependency relation: $\tau_1 \prec \tau_3$. The properties for the tasks is found below:

Task	*bcet*	*wcet*	*d*	*o*	*p*
τ_1	3	3	15	0	15
τ_2	2	2	10	0	10
τ_3	4	4	35	0	35
τ_4	5	5	13	0	13

5.4.6 Multiprocessor example 1

This example is found in [20]. It illustrates a simple multiprocessor system, with two processing elements and one inter-processor dependency ($\tau_2 \prec \tau_3$). The system is not schedulable using Rate Monotonic scheduling. But can be scheduled using Earliest Deadline First. The tasks is defined as follows:

Task	*bcet*	*wcet*	*d*	*o*	*p*	pe
τ_1	2	2	4	0	4	1
τ_2	2	2	6	0	6	1
τ_3	2	2	6	0	6	2
τ_4	3	3	6	0	6	2

5.4.7 Multiprocessor example 2

This system is again based on [20], but this time a communication is added between the two processing elements. This is done according to [12], where a message task is added between the two processing elements. By introducing the message task: τ_m the dependency relation in this example will be: $\tau_2 \prec \tau_m \prec \tau_3$. The tasks is defined as follows:

Task	*bcet*	*wcet*	*d*	*o*	*p*	pe
τ_1	2	2	4	0	4	1
τ_2	1	1	6	0	6	1
τ_3	2	2	6	0	6	2
τ_4	3	3	6	0	6	2
τ_m	1	1	6	0	6	m

Once again the system is schedulable using Earliest Deadline First, but not using Rate Monotonic scheduling.

5.4.8 Multiprocessor example 3

The system consists of 10 tasks on 6 different processing elements, with a inter-processor dependency. The parameters of the tasks can be found below:

Task	*bcet*	*wcet*	d	o	p	pe
τ_1	2	2	40	0	40	1
τ_2	2	2	40	0	40	1
τ_3	2	2	40	0	40	2
τ_4	2	2	40	0	40	2
τ_5	2	2	40	0	40	3
τ_6	2	2	40	0	40	3
τ_7	2	2	40	0	40	4
τ_8	2	2	40	0	40	4
τ_9	2	2	40	0	40	5
τ_{10}	2	2	40	0	40	6

An inter-processor dependency: $\tau_2 \prec \tau_3$ is introduced as well. In the model proposed by A. Brekling, the above example could be verified in under two minutes on a 3.4 GHz PC with 1GB of Ram. In MoVES this example was verified on a 3.4 GHz PC with 2GB of ram in only 1 second.

5.5 Summary

Each feature, such as the allocation protocols, have been tested individually and the produced schedules has been investigated manually in order to ensure correct performance. All cases of this test performed as expected and the implementation in UPPAAL is therefore assumed to be done correctly according to the described MPSoC model.

The tool has been tested on a system containing several realistic size applications from a smart phone. It has been shown that the tool is able to handle this example if it is assumed that tasks only run in worst case execution time.

The maximum size of systems that can be verified has been found. The results shows, that making the model-checker decide when to stop (instead of the Finish state as explained in section 3.4) increases performance a bit (see Figure 5.16. This supports our assumption that the static upper bound is indeed the worst case number of cycles that needs to be checked for a missed deadline, even

though we do not have a formal proof.

The nondeterminism introduced when best and worst case execution times are not the same, results in a state explosion in the model checking. This makes it infeasible to verify systems of realistic sizes when best and worst case execution times are different.

In comparison with the model proposed in [4], the tool developed in this thesis, is able to handle system of a magnitude that is 100 times larger.

Chapter 6

Future Work

Having developed the MoVES tool build upon the formal semantics in UPPAAL, proposed future extensions are divided into two parts. Firstly extending the MPSoC model with more features making it even more realistic and useful to designers, and secondly adding a graphical user interface which gives a better overview of an MPSoC system.

6.1 MPSoC features

First a few simple extensions, which were left out because of time limitations is presented. Later more radical ideas are presented, which requires more thorough work to realize.

6.1.1 Synchronization model

The synchronizer is modelled only to handle **and** dependencies in the sense that if $\tau_y \prec \tau_z, \tau_x \prec \tau_z$ both τ_x **and** τ_y must finish before τ_z can start. Another type of dependency would be to allow τ_z to start as long as either τ_x **or** τ_y has finished. As an example this could be used for a processing element waiting

for data from one or more sensors. This could quite easily be implemented by changing the synchronizer module and define the type of dependency for each dependency.
Handling conditional branching in the task graph is an other possibility. Depending on the output of the execution of a task, a task is chosen which has its dependency resolved. This extension would also require the modelling of how branching is decided in order to update dependencies dynamically. Again only the synchronizer module is affected by this extension, and could fairly simple be changed to include conditional branching.

6.1.2 Hierarchial scheduling

In our MPSoC model only one level of scheduling is handled. Hierarchical (or multi-level) schedulers generalize the traditional role of schedulers (i.e., scheduling threads or processes, in our model tasks) by allowing them to allocate execution time to other schedulers [16]. At the lowest level of the hierarchy, the tasks is scheduled as normal. The scheduled tasks at this level is then scheduled with tasks one level above in the hierarchy. This process is continued until the root scheduler is reached, and a single task is reached.

Extending the UPPAAL semantics to include a fixed number of scheduling levels, definitely seems feasible. Consider a structure with M different sub schedulers and N task in each. This structure together with a list of M scheduling principles and a top scheduling principle would specify the hierarchy fully. The scheduler could fairly easy be changed to schedule tasks on two levels using these structures.

Generalizing this model, a tree structure consisting of sub-schedulers and their tasks should be build. This could be implemented using arrays for the tree structure and the scheduler could be changed to handle this without changing the rest of the model.

6.1.3 Resource usage in part of task execution

In our model power usage is constant throughout the entire execution time of a task. A more realistic model is given in [22], where the power consumption of a task is variable as a function of the processing element frequency. As the concept of processing frequency is already included in our model, a function for automatically calculating the power consumption could be included in the model fairly easy.

As the model works now, it is not possible to specify different costs or resource use in different execution cycles. If a task is executing and uses a resource, it will use the resource the entire runtime. This simplifies the task template a lot. There are two intuitive issues for modelling resource usage in parts of execution times. If a task needs a resource later than the start of its execution time it must send a request to the resource allocator(operating system). If the resource is free, the task will continue running. If the resource is used, the task will be preempted. The task should also issue a release signal to the operating system, if it releases a resource but continues running. In this way, resources will need to be handled in a request/access way. Both the operating system and the task model would have to be changed in order to include the new concepts of request/access and release of resources.

6.1.4 Online mapping of tasks

So far the model assumes, that tasks are mapped statically onto processing elements. Removing this limitation and allow tasks to migrate in real time, would make systems more schedulable, as the available resources are optimized in run time. Resources could for example be spared at times where large parts of the system is idle by grouping tasks together on fewer processing elements. This is especially true if the time needed to move the task is small compared to the execution time of the task. Introducing dynamic task migration in the model is definitely of interest, but some fundamental issues will have to be addressed.

It must be decided how tasks should arrive to the system. Do they arrive locally or globally? In the latter case some global resource needs to schedule arriving tasks. If tasks can migrate once they have been mapped, we need to include this in the operating system as well. Along with the normal synchronize, allocate and schedule phases, a migration phase has to be implemented in the operating system. Algorithms for the online mapping and migration must be devised, and it must be decided if these algorithms are modelled to run in zero time.

6.1.5 Resource driven model

The communication in the model is handled by a number of busses that allows only one communication at the time. However a real on-chip network has not been modelled so far. Our idea to model an on-chip network, is to include it in a general resource model. This general resource model, would include processing

elements, network components (routers and wires) and other resources, such as shared memory and external units. In this way the model will contain the concept of global resources. The paradigm of modelling the platform as a set of inter-connected resources, seem to involve quite a comprehensive amount of work, because it would change the model radically, and has been proposed as a ph.d. project by itself.

6.1.6 Operating system in non-zero time

In scheduling theory it is common to model the operating system as running in zero time as done in our model. This assumption is often reasonable as the number of execution cycles of most realistic tasks, are large compared to the operating system overhead. This assumption becomes less reasonable when tasks has short execution times and preemption becomes more likely, forcing the operating system to run more often. In order to capture the exact schedulability of such systems, the overhead time of the operating system should be considered in the model. This becomes even more relevant when tasks are allowed to migrate between processing elements, as the overhead for transfer code should be considered as well. An idea for this would be to introduce intermediate states (between Idle, Ready and Running) in the task template where the time taken by the operating system is modelled to progess. This extension is not trivial as timing is one of the main issues in the UPPAAL and both the task model and the operating system model would need to be changed.

6.1.7 Asynchronous processing elements

Our model is asynchronous in the way that processing elements may run at different frequencies. However in order to ensure the same correct global schedule at all times, the operating systems synchronize between time units, when finish and ready signals are received. This is done in order to ensure a global correct schedule at all times.

The concept of sending reschedule signals to other operating systems, when dependencies are resolved, is necessary in order to notify another operating system, that it might not be running the task with highest priority. The synchronization between operating systems could however be removed giving a more realistic model. This would result in a different semantics giving non deterministic schedules in some special cases as described in Section 2.2.4.2.

6.1.8 Model checking

The memory usage of the model checker is the current bottleneck of the system, in terms of number of tasks, processing elements and the difference between tasks best and worst case execution times. Improving the model checker in terms of less memory usage would increase the size of systems that can feasibly be verified.

A time vs. memory trade off is also interesting to investigate. A way of using less memory could be to use extra space on the hard drive. This way the natural limit of 2 GB memory could be extended, on the expense of more time demanding read and write operations on the disk.

In the case where bcet=wcet, the special verifier developed by J. Illum is used. This version does not store any visited states in order to limit the memory usage. This however makes it impossible to calculate a schedule of the tasks, because it is based upon the trace of states visited by the model checker, as described in Section 3.4. A way of overcoming this problem would be to implement some kind of print function in the UPPAAL tool. Hereby simply dump information of visited states for each task, during verification.

6.2 MoVES tool

As the overall strategy of our research has been to develop models and tools which aids designers of embedded systems, it would be desirable to interface with the tool through an easy-to-use graphical user interface (GUI). The GUI would especially be useful in providing an overview of dependencies and mapping of tasks onto processing elements. So far the schedules provided by our tool are precise, but not very user friendly. A graphical user interface would also give the possibility of presenting schedules in a more understandable way. The construction of a GUI for the MoVES tool is currently running as a *fagpakke-projekt* under the supervision of A. Brekling at IMM-DTU. At the printing time of this thesis, the project has not finished, but has shown promising results in interfacing with the tool provided by this thesis.

Chapter 7

Conclusion

This thesis has presented a tool for formal verification of Multi-Processor Systems-on-Chip. The tool is based on a formal model using timed automata in UPPAAL. Using model checking, properties such as schedulability, memory usage and power consumption can be verified. Using the tool, designers of embedded system are able to evaluate their design, before synthesizing in hardware. It is, in this way, possible to discover bugs earlier in the design process.

The thesis relates to the work done by A. Brekling in [4], who implemented a basic MPSoC model, capturing the concepts of dynamic scheduling and inter-processor dependencies. Our thesis has answered the objectives stated in section 1.3 and the main contributions of our work has been:

- A more realistic model for application and execution platform, by extending the features which the model is able to handle with:
  - Resource allocation as part of the real-time operating system.
  - Non-periodic tasks (sporadic and aperiodic).
  - Cost model for investigating memory usage and power consumption pr. processing elements and on the total execution platform.
  - Deadline types (hard, firm, soft).
  - Inter-processor communication via busses.
  - Non-deterministic[1] execution times of tasks, in an interval between a best and worst case.

[1] The execution times of sequential code is not in fact non-deterministic, but depends on the input variables. However at the task level, the input variables are abstracted away and the execution times are modelled as non-deterministic.

  - Different clock frequencies on processing elements.

- Implement the timed-automata model in an efficient way, pushing the size limit of systems which can be verified. We are thus able to handle MPSoC systems of realistic sizes. It has been shown that various applications from a smart phone (114 tasks total) can be modelled and verified within 2.5 hours using a modified model checker developed by J. Illum at AAU.

- A tool, which wraps the developed timed-automata model into an easy to use program. In addition to verifying properties such as schedulability and resource usage, the tool is able to provide a schedule helping designers understanding the possible short comings of a design.

A clear division between application and execution platform has been made. The timing properties of the application (period, deadline, offset) is defined in seconds and the execution time of the application on the execution platform is given in cycles. The period, deadline and offset is logically connected to the execution cycles through the speed of the processor.

The model has been debugged and tested using small made up cases, known examples from the literature (e.g. [20]) and a case study of the applications from a smart phone. Furthermore a size test, of systems which can be feasibly verified, is included.

The idea of using formal models for verification of MPSoCs on system-level is new. The work done in [4] showed the possibility of formal modelling and verification using timed automata. The goal of our thesis has been to develop a model which is applicable to real life problems, in terms of captured concepts and the size of systems. The smart phone example shows that this goal is reached. In the previous chapter, ideas of future development is presented, which will make the model even more realistic in terms of handled feature and feasible sizes of systems. Especially online task mapping is of interest as the system would be able to better adapt to unforseen events. Resources could for example be spared at times where large parts of the system is idle by grouping tasks together on fewer processing elements.

At the DATE'07 conference, we presented the work of this thesis in the University Booth forum. The response from fellow students and industry confirmed our belief that, formal verification using model checking is of high interest to companies in areas such as automotive and communications industries.

Appendix A

Terms

Task	τ
Processing element	pe
Real-time operating system	os
Processor frequency	f
Dependency relation	$\prec$
Application	A
Execution Platform	E
Set of tasks	T
Set of processing elements	PE
Mapping	$m : T \rightarrow PE$

Task:	
Period	π
Offset	ω
Deadline	δ

Task mapped to a pe	
Best case execution time	bcϵ
Worst case execution time	wcϵ
Period	p
Offset	o
Deadline	d
Dynamic memory	dm
Static memory	sm
Power consumption	pw

Appendix B

Introduction to timed automata

This appendix is written by A. Brekling and was originally included in [4], except for part 7 and 8 which have been written for the purpose of this thesis.

This introduction to timed automata is inspired by lectures and lecture notes on Process and Data Modelling by Jens Christian Godskesen, IT University of Copenhagen and on Real-Time Systems by Jesper Blak Møller, Technical University of Denmark.

In the following, it is assumed that the reader has general knowledge of *finite automata*.

B.1 Finite automata

An example of an electric door with sensors will be used to exemplify the extensions to finite automata.

We define a *finite automaton* M as a five tuple:

$$M = (Q, \Sigma, \rightarrow, q_0, F)$$

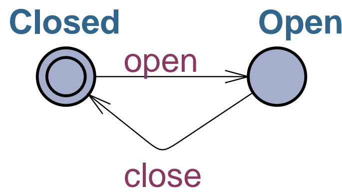

Figure B.1: A finite automaton for the electric door

where

- Q is a set of *states*
- Σ is a finite set of *input symbols*
- $\rightarrow \subseteq Q \times \sigma \times Q$ is the *transition relation*
- q_0 is the *initial state*
- $F \subseteq Q$ is the set of accepting states

If $(q, a, q') \in \rightarrow$ we say that there is a transition from q to q' written as $q \xrightarrow{a} q'$

Door example - finite automata

Consider an electric door. When in the *Closed* state and given an `open` symbol, it moves to the *Open* state.When in the *Closed* state and given a `close` symbol, it moves to the *Closed* state. A model for the door as a finite automata can be seen in Figure B.1:

B.2 Communicating finite automata

A communicating finite automaton is a finite automaton where each transition is either an output, input or internal transition.

We write:

- $q \xrightarrow{a!} q'$ for an output transition,

- $q \xrightarrow{a?} q'$ for an input transition and
- $q \longrightarrow q'$ for an internal transition.

A communicating finite automaton is defined by (the set of accepting states is omitted from now on as we focus on communication/synchronization only):

$$M = (Q, \Sigma, \rightarrow, q_0)$$

where Q, Σ and q_0 are as before and:

$$\rightarrow \subseteq (Q \times \Sigma \times \{!, ?\} \times Q) \cup (Q \times Q)$$

Communicating finite automata run in parallel and synchronize on input/output symbols.

Let $M_1 = (Q_1, \Sigma, \rightarrow_1, q_1)$ and $M_2 = (Q_2, \Sigma, \rightarrow_2, q_2)$ be communicating finite automata. The composition of M_1 and M_2 is the finite automaton:

$$M = (Q_1 \times Q_2, \Sigma, \rightarrow, (q_1, q_2))$$

where $\rightarrow$ is defined by:

$$\frac{p \xrightarrow{a!}_1 p' \quad q \xrightarrow{a?}_2 q'}{(p, q) \xrightarrow{a} (p', q')} \qquad \frac{q \xrightarrow{a!}_2 q' \quad p \xrightarrow{a?}_1 p'}{(p, q) \xrightarrow{a} (p', q')}$$

$$\frac{p \longrightarrow_1 p'}{(p, q) \longrightarrow (p', q)} \qquad \frac{q \longrightarrow_2 q'}{(p, q) \longrightarrow (p, q')}$$

Door example - communicating finite automata

In the electric door example, it is wished to model a sensor as well as the door. The sensor can either have light or no light, modelled by the states *Light* and *NoLight*. When moving from *Light* to *NoLight* an `Obstruct` signal is issued, and when moving from *NoLight* to *Light* an `UnObstruct` signal is issued. The sensor is modelled by the communicating finite automata in Figure B.2. The door is modelled in the communicating finite automaton in Figure B.3. The channels `Obstruct` and `UnObstruct` model whether or not the sensor is obstructed.

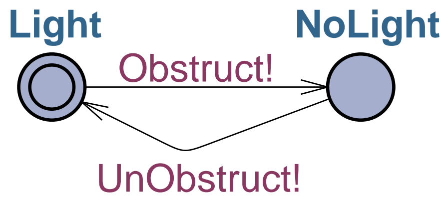

Figure B.2: A communicating finite automaton for the sensor

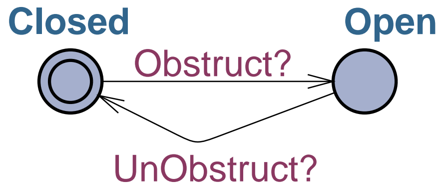

Figure B.3: A communicating finite automaton for the door

B.3 Extended finite automata

A communicating finite automaton may be extended with variables. Execution of a transition may depend on the value of the variables, and values of variables may be assigned when performing a transition.

$$p \xrightarrow{a!, x>10, y:=5} p'$$

means that the transition from p to p' can only be performed if the value of x is greater than 10 (the boolean expression $x > 10$ is called a guard), and when performing the transition the value of y is set to 5.

Door example - extended finite automata

We wish to model that the door cannot handle more than 100 openings and closings per day; this is done by extending the model for the door with the variable `count`. It is increased every time the door opens, and the door cannot open unless `count` is less than 100. It is thought to be reset every day. The

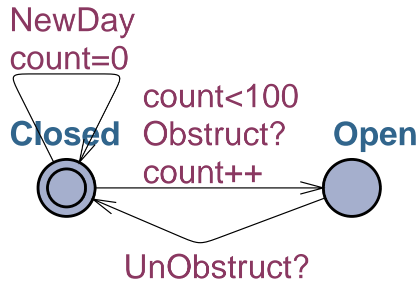

Figure B.4: An extended finite automaton for the door

extended finite automata for the door can be seen in Figure B.4

B.4 Timed finite automata

A timed communicating finite automaton is an extended finite automaton containing real valued clocks. Clocks measure the progress of time and may be part of a guard. The transition:

$$p \xrightarrow{a!, c>10, c:=0} p'$$

can only be performed when the time of clock c is greater than 10, and when the transition is performed the clock is reset. Furthermore, clocks allow for timed invariants on states. A timed invariant specifies when it is possible to be in the state in question. For example, the invariant: $c \leq 10$ specifies that the state cannot be entered unless the clock c is less than or equal to 10. Furthermore, when in the state in question, it must leave before the c exceeds 10.

Door example - timed finite automata

We wish to model that the door is kept open at least 5 seconds after every opening. If the sensor gets obstructed within those 5 seconds, the door must be kept open for another 5 seconds. Therefore, the local clock `x` is introduced in the model. Resetting the variable `count` each day is done by introducing a clock `day`, which, every 86400 seconds, resets `count` and itself in any state of the model. The model for the door as a timed automaton can be seen in Figure B.5.

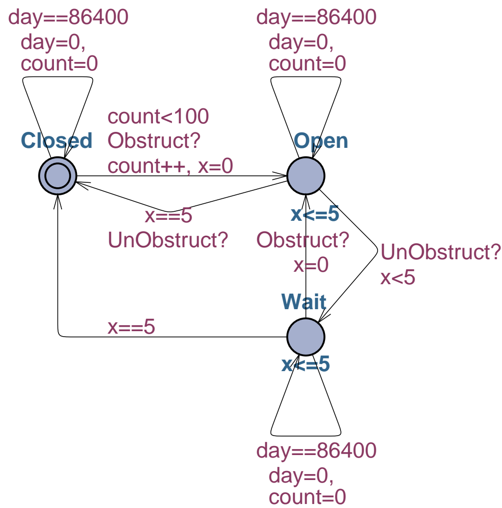

Figure B.5: A timed automaton for the door

B.5 UPPAAL

UPPAAL is a tool with which timed communicating finite automata can be modelled using a graphical editor. Essentially, an UPPAAL model contains:

- global declarations of channels, variables and clocks
- a parameterized template for each type of automaton including local declarations of variables and clocks
- a definition of each automaton based on its template
- a system definition consisting of the automata making up the system

With a system declaration, UPPAAL provides a tool for simulating the system and a tool for verification of properties of the system. The simulation can be conducted either by choosing a random simulation or manually selecting each transition or a combination of both. In a random simulation, UPPAAL randomly chooses the next transition from the different enabled transitions. In a manual simulation, the user can choose each transition from a list of enabled transitions. If we would like to validate whether a certain property is guaranteed by the system, we must (more or less systematically) try to select all possible

paths through the simulation. In real life models it may be an almost impossible task through simulation to make sure that a certain property of a composed system is satisfied. With the help of the verification tool in UPPAAL, we can have the states of a model explored automatically.

Urgent and committed states

UPPAAL provides convenient notions of urgent and committed states. These are explained here:

Urgent An urgent state is a state where time cannot pass. This could also be modelled as follows: a local clock x is reset to 0 on all incoming transitions to the state, and an invariant $x \leq 0$ forces exit of the state before time passes.

Committed Committed state are - like urgent - states where time cannot pass. Furthermore, when at least one timed automaton of a system is in a committed state, only transitions leaving committed states are enabled.

B.6 Model checking

Verification in UPPAAL is done using model checking. Basically, model checking explores all possible behaviors. A model M is created in a formalism accepted by the model checking tool (e.g. finite state machine), requirements R are specified in temporal logic, and finally all states of M are found and R is checked in each state [7].

In UPPAAL, the specification language for requirements is a subset of timed computation tree logic. Requirements are expressed in terms of formulae that refer to the computation tree of the system. In Figure B.6 an example of a computation tree is given.

UPPAAL's specification language has two types of path and state quantifiers:

E Exists a path

A For all paths

G All states in path ([] in UPPAAL)

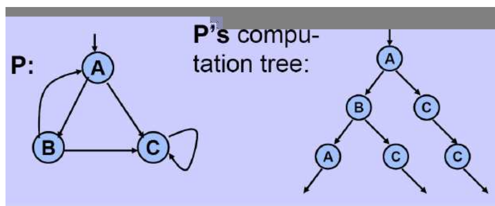

Figure B.6: Example of a computation tree

F Some state in path (<> in UPPAAL)

The standard operators UPPALL allows are:

1. `E<> p`: For some path, p holds in some state (p holds sometime)
2. `A[] p`: For all paths, p holds in all states (p holds invariantly)
3. `E[] p`: For some path, p holds in all states (p holds potentially always)
4. `A<> p`: For all paths, p holds in some state (p holds inevitably)

Door example - model checking

If we want to verify that the door cannot be opened more than 100 times, this can be expressed in UPPAAL by the following query:

$$A[]D.count < 101$$

And the response from the UPPAAL model checker is `Property is satisfied`

If we wanted to see whether it was possible to open the door 100 times we could instead check the following query:

$$E <> D.count == 100$$

Again the response from the UPPAAL model checker is `Property is satisfied`. If the option *Diagnostic Trace* is checked, the simulator will then provide a trace for a path where the property holds in some state.

B.7 Priorities

When model checking a timed automata model i UPPAAL the ability for state explosion occurs. This depends on the number of enabled transitions in each state. If two templates is able to take a transition the same time, it will result in two different branches in the verification tree. E.g. τ_1 and τ_2 is enabled at the same time, then one branch will be with τ_1 taking a transition before τ_2 and vica versa. These two branches might lead to the same result in the end. Therefore UPPAAL introduces priorities, these are defined as: $\tau_1 < \tau_2$. This will make UPPAAL always take the transition of τ_2 before τ_1. This will reduce the size of the computation tree. The priorities are assigned when defining a system in UPPAAL.

B.8 Select

So far, only deterministic transitions have been considered, meaning that when taking a transition, the result will not differ. However, UPPAAL offers a property for each transitions, called: *Select*. The select property, non-deterministically binds a given identifier to a value in a given range. E.g. if the select statement is defined as: `i:int[1,2]` then two transitions will be available for the modelchecker. One will lead to: i=1 and the other to: i=2. However this propery will increase the size of the computation tree, since every time there is a non-deterministic select, each branch will result in a different branch of the computation tree.

Appendix C

Using MoVES

MoVES is a tool for modelling and verification of embedded systems. MoVES is based on timed automata, and uses UPPAALs verifier: Verifyta.

The tool is currently working with Java J2SE 1.5.0 on a windows XP machine.

C.1 Defining a system for MoVES

Defining a system for MoVES is done in: `MPSoC.java`. This file contains information about the entire MPSoC system. Definitions of tasks, processing elements and mapping of these. Description on defining a simple system is given below.

A complete example with four tasks mapped onto two processing elements connected by a bus. One inter-processor dependency with a message task and costs can be found in Listing C.1.

C.1.1 Application

First the application must be defined. This is done by creating a number of tasks, costs and dependencies. A task t is defined as below:

```
Task t = new Task(bcet, wcet, deadline, offset, period, fixed
    priority);
```

A task is defined by: best and worst case execution times (bcet, wcet), deadline in seconds, offset in seconds, period in seconds and a fixed priority.

Another kind of task, is a triggered non-periodic task. These tasks are defined as above but with fewer arguments:

```
Task t = new Task(bcet, wcet, deadline, fixed priority);
```

The triggered tasks can be issued in two ways. Either by having its dependency resolved, and hereby be activated by the controller, or an environment. The environment will be defined in the next section.

The deadline of a task can be of the following kind: hard, firm and soft. If a task with a hard deadline misses, the system will deadlock. A task with a firm deadline stop execution if its deadline is missed. A task with a soft deadline will continue running until it finishes, and hereafter start over. All tasks is by default hard deadlined, but can be set to either soft or firm with one of the following commands:

```
t.softDeadline();
t.firmDeadline();
```

Note that, if any task is defined as soft deadlined, the property: `allFinish()` can not be used during verification, because the period of the task is skewed incresing the time after which the schedule repeats.

C.1.2 Platform

Having defined the desired number of tasks, the platform must be specified. This is done by defining a number of processing elements.

```
Processor p = new Processor(frequency, scheduling algorithm,
    allocation algorithm);
```

The frequency is defined in Hz.
The scheduling algorithm is defined by using:

Processor.RM Rate Monotonic Scheduling

Processor.DM Deadline Monotonic Scheduling

Processor.FP Fixed Priority Scheduling

Processor.EDF Earliest Deadline First Scheduling

Along with the scheduling algorithms which are defined as:

Processor.PSC Preemptive Critical Section

Processor.NPSC Non Preemptive Critical Section

Processor.PRI_INH Priority Inheritance

If there are any inter-processor dependencies, a bus must be defined for this communication. A bus is defined as a normal processing element. Once a communication has started on the bus it cannot be interrupted. This is done using the same resource for all message tasks running on the same bus, hereby making the message task non-preemptable. A resource, *bus*, is defined as:

```
Resource bus = new Resource();
```

Finally all processing elements must be mapped onto the platform, by putting them in a processor array. This is done as below:

```
Processor[] ps = {p1,p2,...,pm}
```

If a triggered task, *t*, is set to be triggered by an environment *e*, it can be defined as follows:

```
Environment e = new Environment(t, interarrival time);
```

The minimum inter-arrival time is defined in seconds.

C.1.3 Mapping

The application implementation is defined by mapping tasks onto processing elements. The tasks is mapped onto processing elements in a double-array `tasks`.

```
Task[][] tasks = {{t1, t2},{t3, t4},{tm}};
```

The first dimension of the array is the processor number and the second dimension contains all tasks mapped onto it.
Now that all tasks and processing elements are defined, and the mapping is set, it should all be collected in application and platform. For this purpose there are a couple of lines which may not be omitted from the MPSoC.java file, these are:

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

pl = new Platform(ps);
apps = new Application(tasks, ca, granularity);
```

If there exists inter-processor dependencies in the application, this is modeled in the system, using message tasks. The message task is defined as a trigger task, which has best and worst case execution times, deadline and fixed priority.

```
Task tm  = new Task(bcet, wcet, deadline, fixed priority);
```

When the communication takes place, it is crucial that it can not be preempted. For this purpose a resource can be used. By setting all message-tasks to use the same resource, they cannot preempt each other, no matter their priorities. A resource usage, of resource *bus*, for a message task is defined in the same way as other tasks as seen below.

```
apps.useResource(tm,bus);
```

A dependencies between $\tau_1 \prec \tau_2$ is modelled using the following command:

```
apps.addDep(t2,t3);
```

C.1.4 Costs

Costs can be added to the different tasks. Default costs are: Memory and Power, but these can easily be extended by adding additional costs to the cost array *ca* described above.
A cost for a task is defined as: static cost and dynamic cost (in states: idle, ready, running and preempted). The costs for a tasks is defined as below:

```
memory.use(t, static, idle, ready, running, preempted);
```

Each use of costs should be defined in the same unit(MB, Byte, etc.).

Two tasks can share a cost. This could be results of calculations which are stored for another task etc. This is shown below:

```
memory.share(t1, t2, 5);
```

This means that the tasks: t_1 shares 5 units of memory with t_2. By setting this shared cost, the memory will not be deleted before t_2 has begun its execution.

C.2 Running MoVES

Once the system is defined in MPSoC.java, the tool must be compiled with the new MPSoC definition. Done by using the command:

```
javac MOVES.java
```

Before executing the tool, some queries must be declared. For an introduction to queries refer to: C.2.1.

Now that, the system is compiled and the queries are specified, the tool program can be executed. A number of arguments can be used (refer to: C.2.2). The system is executed using the following command:

```
java MOVES [args]
```

and prints the results to teh screen.

C.2.1 Specifying queries for MoVES

The queries for MoVES is specified in the query-lanquage CTL[1]. The queries are specified in a file (e.g. bool.q) with one query on each line. Queries should be of type: $E <> \varphi$ (Will φ eventually be satisfied) or $A[]\varphi$ (where φ always holds)[2].

The obvious property to check is whether all tasks meet their deadlines. This is checked by query: `E<>missedDeadline` in a query file (e.g. bool.q). If this property is not satisfied (all tasks meet their deadlines). If the user wants a schedule for a schedulable system, another property must be verified: `E<>allFinish()`.

Other properties that can be verified are:

E <> preempted Is any task preempted at some time?

E <> totalCostUsed(costtype)>limit Does a specified cost, added up on all processors, exceed a certain limit?

E <> totalCost[processor][costtype]>limit Does a specified cost exceed a certain limit on a specified processor?

E <> Taskc1.cp > limit Does the response-time of task 1 exceed a certain limit?

C.2.2 Arguments for MoVES

In MoVES there is a number of arguments that can be used when running the program. They are as follows:

-s Defines a list of scheduling algorithms which should be used in the verification. The algorithms are:

- **FP** Fixed priority scheduling
- **RM** Rate Monotonic scheduling
- **DM** Deadline Monotonic scheduling
- **EDF** Earliest Deadline First scheduling

-a All mappings will be verified

[1]Computational tree logic
[2]Where: $A[]\varphi = \neg E <> \neg\varphi$

-f X Here X specifies a queryfile which should be used in the verification. If ommitted the file `bool.q` is used.

-tX Here X specifies for how long a schedule should be shown (E.g the argument -t10 will give a schedule for 10 timeunits). If X is ommitted a schedule for the first two hyperperiods is shown.

-e Shows the notation of a schedule.

- **0** The task is not running.
- **1** The task is running.
- **-** The task is in its offset.
- **x** The task has missed its deadline.
- **X** The task has missed its deadline and is running.
- ***** The task has finished 2x the hyperperiod.

-o X Saves the UPPAAL-system in the file named X.xml.

-g X Where X specifies a granularity for the system used in the verification.

-u Writes the utilisation for each processor.

-h, -?, -help Shows the help file for the system.

Listing C.1: Small example for defining MPSoC.java

```
public class MPSoC {
    public Application apps;
    public Platform pl;

    public MPSoC(int granularity) {
        Resource bus = new Resource();
        //Defines tasks (Execution time, 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, 5);

        //Defines processors
        Processor p1 = new Processor(1, Processor.RM, Processor.NPCS);
        Processor p2 = new Processor(1, Processor.EDF, Processor.NPCS);
        Processor pm = new Processor(1, Processor.RM, Processor.NPCS);

        //Assigns tasks to processors
        Task[][] tasks = {{t1,t2},{t3,t4},{tm}};

        //Adds the processors to the system
        Processor[] ps = {p1,p2,pm};

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

        pl = new Platform(ps);
        apps = new Application(tasks, ca, granularity);

        //Use the resource for the message task
        apps.useResource(tm,bus);

        //Assign memory usage
        memory.set(t1,1,0,0,3,3);
        memory.set(t2,1,0,0,5,5);
        memory.set(t3,2,0,0,6,6);
        memory.set(t4,1,0,0,9,9);
        memory.share(t2,t3,5);
        power.set(t1,0,0,0,5,0);
        power.set(t2,0,0,0,5,0);
        power.set(t3,0,0,0,10,0);
        power.set(t4,0,0,0,10,0);

        //Adds dependencies to the system.
        //The dependencies references to the name of the task-object
        apps.addDep(t2,tm);
        apps.addDep(tm,t3);
    }
}
```

Appendix D

Evaluation Results

This appendix contains the results of the different tests described in chapter: 5.

Highest priority first

RM-scheduling

```
Verification time: 0.00 min
E<>allFinish(): true
            5   10   15   20   25   30   35   40
Task: 1 1000100010001000100010001000100010001000
Task: 2 0110011000110001010001100110001100010100
```

DM-scheduling

```
Verification time: 0.00 min
E<>allFinish(): true
            5   10   15   20   25   30   35   40
Task: 1 0010100010001000010000101000100010000100
Task: 2 1100011000110001100011000110001100011000
```

FP-scheduling

```
Verification time: 0.00 min
E<>allFinish(): true
            5    10    15    20    25    30    35    40
Task: 1 1000100010001000100010001000100010001000
Task: 2 0110011000110001010001100110001100010100
```

EDF-scheduling

```
Verification time: 0.00 min
E<>allFinish(): true
            5    10    15    20    25    30    35    40
Task: 1 0010100010001000010000101000100010000100
Task: 2 1100011000110001100011000110001100011000
```

Finish at end of period

```
Verification time: 0.00 min
E<>allFinish(): true
            5
Task: 1 11111111
```

Highest priority with dependency

Interprocessor dependency

```
Verification time: 0.00 min
E<>allFinish(): true
            5    10
Task: 1 001100001100
Task: 2 110000110000
```

Intraprocessor dependency

```
Verification time: 0.00 min
E<>allFinish(): true
            5    10
Task: 1 001100001100
Task: 2 110000110000
```

Ready depends on task with offset

Inter-processor dependency

```
Verification time: 0.00 min
E<>allFinish(): true
            5    10   15   20   25   30
Task: 1 000011001100110000001100110000
Task: 2 --1100110011001100110011001100
```

Intra-processor dependency

```
Verification time: 0.00 min
E<>allFinish(): true
            5    10   15   20   25   30
Task: 1 000011001100110000001100110000
Task: 2 --1100110011001100110011001100
```

Ready and finish at same time

Inter-processor dependency

```
Verification time: 0.00 min
E<>allFinish(): true
            5    10   15
Task: 1 110011001100**
Task: 2 --110011001100
```

Intra-processor dependency

```
Verification time: 0.00 min
E<>allFinish(): true
            5    10   15
Task: 1 110011001100**
Task: 2 --110011001100
```

Chain of dependencies to resolve

Inter-processor dependency

```
Verification time: 0.00 min
E<>allFinish(): true
            5   10
Task: 1 110000110000
Task: 2 001100001100
Task: 3 000011000011
```

Intra-processor dependency

```
Verification time: 0.00 min
E<>allFinish(): true
            5   10
Task: 1 110000110000
Task: 2 001100001100
Task: 3 000011000011
```

Four tasks depends on one task

Inter-processor dependency

```
Verification time: 0.02 min
E<>allFinish(): true
            5   10   15   20
Task: 1 11000000001100000000
Task: 2 00110000000011000000
Task: 3 00110000000011000000
Task: 4 00110000000011000000
Task: 5 00110000000011000000
```

Intra-processor dependency

```
Verification time: 0.01 min
E<>allFinish(): true
            5   10   15   20
Task: 1 11000000001100000000
Task: 2 00110000000011000000
Task: 3 00001100000000110000
Task: 4 00000011000000001100
Task: 5 00000000110000000011
```

One task depends on four other tasks

Inter-processor dependencies

```
Verification time: 0.02 min
E<>allFinish(): true
            5   10   15   20
Task: 1 11000000001100000000
Task: 2 00110000000011000000
Task: 3 00110000000011000000
Task: 4 00110000000011000000
Task: 5 00110000000011000000
```

Intra-processor dependencies

```
Verification time: 0.01 min
E<>allFinish(): true
            5   10   15   20
Task: 1 11000000001100000000
Task: 2 00110000000011000000
Task: 3 00001100000000110000
Task: 4 00000011000000001100
Task: 5 00000000110000000011
```

Utilisation

Utilisation equals 1.00

```
Utilization:
Processor 1:    1,00
Verification time: 0.00 min
E<>allFinish(): true
            5   10
Task: 1 111000111000
Task: 2 000111000111
```

Utilisation equals 2.00

```
Utilization:
Processor 1:    2,00
Utilisation higher than 1.00
System not schedulable
```

Several preemptions of same task

```
Verification time: 0.02 min
E<>allFinish(): true
            5   10   15   20   25   30   35   40   45   50   55   60   65   70   75   80   85
Task: 1 --110000000000000000110000000000000000110000000000000000110000000000000000************
Task: 2 ------110000000000000000110000000000000000110000000000000000110000000000000000********
Task: 3 ----------110000000000000000110000000000000000110000000000000000110000000000000000****
Task: 4 --------------110000000000000000110000000000000000110000000000000000110000000000000000
Task: 5 110011001100110011110011001100110011110011001100110011110011001100110011**************
```

Several preemptions of different tasks

```
Verification time: 0.02 min
E<>allFinish(): true
            5   10   15   20   25   30   35   40   45   50   55   60
Task: 1 --------110000000000000000110000000000000000110000000000000000
Task: 2 ------110011000000000000110011000000000000110011000000000000**
Task: 3 ----110000001100000000110000001100000000110000001100000000****
Task: 4 --110000000000110000110000000000110000110000000000110000******
Task: 5 110000000000000000111100000000000000111100000000000000011********
```

Resolved dependency leads to preemption

Inter-processor dependency

```
Verification time: 0.00 min
E<>allFinish(): true
            5   10
Task: 1 110000110000
Task: 2 001100001100
Task: 3 110011110011
```

Share resources, highest priority first

```
Verification time: 0.00 min
E<>allFinish(): true
            5   10   15
Task: 1 1100000011000000
Task: 2 0011000000110000
Task: 3 0000111000001110
```

Resource allocation

Non preemptive critical section

```
Verification time: 0.00 min
E<>allFinish(): true
            5   10   15   20   25   30
Task: 1 --011100000001110000000111000000
Task: 2 -000001110000000111000000011100*
Task: 3 111000000011100000001110000000**
```

Preemptive critical section

```
Verification time: 0.00 min
E<>allFinish(): true
            5   10   15   20   25   30
Task: 1 --000011100000001110000000111000
Task: 2 -111000000011100000001110000000*
Task: 3 100011000010001100001000110000**
```

Priority inheritance

```
Verification time: 0.00 min
E<>allFinish(): true
            5   10   15   20   25   30
Task: 1 --001110000000111000000011100000
Task: 2 -100000110010000011001000001100*
Task: 3 101100000010110000001011000000**
```

Different processor speeds

```
Verification time: 0.00 min
E<>allFinish(): true
            5
Task: 1 10001000
Task: 2 11001100
```

Granularity

```
Verification time: 0.00 min
E<>allFinish(): true
            5   10
```

```
Task: 1 100010001000*
Task: 2 110011001100*
Task: 3 -111011101110
```

Timing anomaly

All finishes

```
Verification time: 0.03 min
E<>allFinish(): true
            5    10   15   20
Task: 1 11100000001110000000
Task: 2 00000011000000001100
Task: 3 11000000001100000000
Task: 4 00010000000001000000
Task: 5 00011100000001110000
Task: 6 00100000000010000000
```

A deadline is missed

```
Verification time: 0.01 min
E<>missedDeadline: true
            5
Task: 1 100000
Task: 2 000011
Task: 3 110000
Task: 4 00000X
Task: 5 011100
Task: 6 000010
```

Granularity prevents timing anomaly

Before granularity - deadlines met

```
E<>allFinish(): true
            5    10   15   20   25
Task: 1 1111000000000111100000000
Task: 2 1111000000000111100000000
Task: 3 0000000011100000000001110
Task: 4 0000001111000000000111100
Task: 5 0000001100000000000110000
Task: 6 0000110000000000011000000
```

Before granularity - deadline missed

```
Verification time: 0.01 min
E<>missedDeadline: true
            5   10
Task: 1 11100000000
Task: 2 11110000000
Task: 3 00000111000
Task: 4 0000000111X
Task: 5 00011000000
Task: 6 00000110000
```

After granularity

```
Verification time: 0.02 min
E<>allFinish(): true
            5   10
Task: 1 110000110000
Task: 2 110000110000
Task: 3 000011000011
Task: 4 000110000110
Task: 5 000100000100
Task: 6 001000001000
```

Softdeadline

```
Verification time: 0.00 min
E<>missedDeadline: false
```

Firm deadline

Deadline equals period, run

```
Verification time: 0.00 min
E<>allFinish(): true
            5   10
Task: 1 111110111110
Task: 2 000001000001
```

Deadline equals period, no run

```
Verification time: 0.00 min
```

```
E<>allFinish(): true
            5    10
Task: 1 111111111111
Task: 2 000000000000
```

Deadline < period, run

```
Verification time: 0.00 min
E<>allFinish(): true
            5    10    15
Task: 1 1111100011111000
Task: 2 0000010000000100
```

Deadline < period, no run

```
Verification time: 0.00 min
E<>allFinish(): true
            5    10    15
Task: 1 1111110011111100
Task: 2 0000000000000000
```

Cost model

Static cost

Case 1

```
Verification time: 0.00 min
E<>totalCostUsed(Memory)<5 && Taskc2.IdleWait: false
E<>totalCostUsed(Memory)<5 && Taskc2.ReadyDynamic: false
E<>totalCostUsed(Memory)<5 && Taskc2.Running3: false
```

Case 2

```
Verification time: 0.00 min
E<>totalCostUsed(Memory)<5 && Taskc2.IdleWait: false
E<>totalCostUsed(Memory)<5 && Taskc2.ReadyDynamic: false
E<>totalCostUsed(Memory)<5 && Taskc2.Running3: false
```

Ready cost

Case 1

```
Verification time: 0.00 min
E<>Taskc2.IdleWait: false
E<>totalCostUsed(Memory)<5 && Taskc2.IdleWait: false
E<>Taskc2.ReadyDynamic: true
E<>totalCostUsed(Memory)<5 && Taskc2.ReadyDynamic: false
E<>Taskc2.Running3: true
E<>totalCostUsed(Memory)<5 && Taskc2.Running3: true
```

Case 2

```
Verification time: 0.00 min
E<>Taskc2.IdleWait: false
E<>totalCostUsed(Memory)<5 && Taskc2.IdleWait: false
E<>Taskc2.ReadyDynamic: false
E<>totalCostUsed(Memory)<5 && Taskc2.ReadyDynamic: false
E<>Taskc2.Running3: true
E<>totalCostUsed(Memory)<5 && Taskc2.Running3: true
```

Running cost

Case 1

```
Verification time: 0.00 min
E<>Taskc2.IdleWait: false
E<>totalCostUsed(Memory)<5 && Taskc2.IdleWait: false
E<>Taskc2.ReadyDynamic: true
E<>totalCostUsed(Memory)<5 && Taskc2.ReadyDynamic: true
E<>Taskc2.Running3: true
E<>totalCostUsed(Memory)<5 && Taskc2.Running3: false
```

Case 2

```
Verification time: 0.00 min
E<>Taskc2.IdleWait: false
E<>totalCostUsed(Memory)<5 && Taskc2.IdleWait: false
E<>Taskc2.ReadyDynamic: false
E<>totalCostUsed(Memory)<5 && Taskc2.ReadyDynamic: false
E<>Taskc2.Running3: true
E<>totalCostUsed(Memory)<5 && Taskc2.Running3: false
```

Idle cost

Case 1

```
Verification time: 0.00 min
E<>totalCostUsed(Memory)<5 && Taskc2.IdleWait: false
E<>Taskc2.IdleWait: true
E<>totalCostUsed(Memory)<5 && Taskc2.ReadyDynamic: true
E<>totalCostUsed(Memory)<5 && Taskc2.Running3: true
```

Case 2

```
Verification time: 0.00 min
E<>totalCostUsed(Memory)<5 && Taskc2.IdleWait: false
E<>Taskc2.IdleWait: false
E<>totalCostUsed(Memory)<5 && Taskc2.ReadyDynamic: true
E<>totalCostUsed(Memory)<5 && Taskc2.Running3: true
```

Preempted Cost

Case 1

```
Verification time: 0.00 min
E<>Taskc2.IdleWait: false
E<>totalCostUsed(Memory)<5 && Taskc2.IdleWait: false
E<>Taskc2.ReadyDynamic: true
E<>totalCostUsed(Memory)<5 && Taskc2.ReadyDynamic: false
E<>Taskc2.Running3: true
E<>totalCostUsed(Memory)<5 && Taskc2.Running3: true
```

Case 2

```
Verification time: 0.00 min
E<>Taskc2.IdleWait: false
E<>totalCostUsed(Memory)<5 && Taskc2.IdleWait: false
E<>Taskc2.ReadyDynamic: false
E<>totalCostUsed(Memory)<5 && Taskc2.ReadyDynamic: false
E<>Taskc2.Running3: true
E<>totalCostUsed(Memory)<5 && Taskc2.Running3: true
```

Shared Cost

Case 1

```
Verification time: 0.00 min
E<>Taskc2.Idle: true
E<>totalCostUsed(Memory)==5 && Taskc2.Idle: true
E<>Taskc2.Ready: true
E<>totalCostUsed(Memory)==5 && Taskc2.Ready: true
E<>Taskc2.Running: true
E<>totalCostUsed(Memory)==5 && Taskc2.Running3: true
```

Case 2

```
Verification time: 0.00 min
E<>Taskc2.Idle: true
E<>totalCostUsed(Memory)==5 && Taskc2.Idle: true
E<>Taskc2.Ready: true
E<>totalCostUsed(Memory)==5 && Taskc2.Ready: true
E<>Taskc2.Running: true
E<>totalCostUsed(Memory)==5 && Taskc2.Running3: true
```

Environment

```
Verification time: 2.44 min
E<>allFinish(): false

Verification time: 0.00 min
E<>missedDeadline: true

Task: 1 01
Task: 2 10
```

Comparison

Single Processor example 1

```
RM
Verification time: 0.00 min
E<>allFinish(): true
E<>missedDeadline: false

EDF
Verification time: 0.00 min
E<>allFinish(): true
E<>missedDeadline: false
```

Single Processor example 2

```
RM
Verification time: 0.00 min
E<>allFinish(): false
E<>missedDeadline: true

EDF
Verification time: 0.00 min
E<>allFinish(): true
E<>missedDeadline: false
```

Single Processor example 3

```
RM
Utilisation higher than 1.00
System not schedulable

EDF
Utilisation higher than 1.00
System not schedulable
```

Single Processor example 4

```
RM
Verification time: 0.00 min
E<>allFinish(): false
E<>missedDeadline: true

C:\Documents and Settings\Je
MOVES -s DM
DM
Verification time: 0.00 min
E<>allFinish(): true
E<>missedDeadline: false
```

Single Processor example 5

```
RM
Verification time: 0.00 min
E<>allFinish(): false
E<>missedDeadline: true

EDF
Verification time: 0.02 min
E<>allFinish(): true
E<>missedDeadline: false
```

Multi Processor example 1

```
RM, RM
Verification time: 0.00 min
E<>allFinish(): false
E<>missedDeadline: true
```

```
EDF, EDF
Verification time: 0.00 min
E<>allFinish(): true
E<>missedDeadline: false
```

Multi Processor example 2

```
RM, RM
Verification time: 0.00 min
E<>allFinish(): false
E<>missedDeadline: true
```

```
EDF, EDF
Verification time: 0.00 min
E<>allFinish(): true
E<>missedDeadline: false
```

Multi Processor example 3

```
RM, RM, RM, RM, RM, RM
Verification time: 0.00 min
E<>allFinish(): true
E<>missedDeadline: false
```

```
EDF, EDF, EDF, EDF, EDF, EDF
Verification time: 0.00 min
E<>allFinish(): true
E<>missedDeadline: false
```

```
FP, FP, FP, FP, FP, FP
Verification time: 0.00 min
E<>allFinish(): true
E<>missedDeadline: false
```

```
DM, DM, DM, DM, DM, DM
Verification time: 0.00 min
E<>allFinish(): true
E<>missedDeadline: false
```

Appendix E

Java code

E.1 MoVES.java

```
/**
* MOVES.java
* Main class for MoVES.
* The MoVES class handles inputs writtes by the user in the command-line
*
* Generates a MPSoC-object and Verifier-object which handles the verification.
*
* @author    Kristian Staaloe Knudsen, Jens Ellebaek Nielsen
* @version   1.1
* @access    public
*/
package MoVES;

import java.lang.*;
import java.io.*;

public class MoVES {
    /**
     * Main class for MoVES.
     * @param args String[] containing the following commands
     * -s Defines a list of scheduling algorithms which should be used in the
          verification.<br>
     * The algorithms are:<br>
     * FP Fixed priority scheduling<br>
     * RM Rate Monotonic scheduling<br>
     * DM Deadline Monotonic scheduling<br>
     * EDF Earliest Deadline First scheduling<br>
     * -a All mappings will be verified<br>
     * -f X Here X specifies a queryfile which should be used in the verification.
          If ommitted the file bool.q is used.<br>
     * -tX Here X specifies for how long a schedule should be shown<br>
     * -e Shows the notation of a schedule.<br>
     * -o X Saves the UPPAAL-system in the file named X.xml.<br>
     * -g X Where X specifies a granularity for the system used in the
          verification.<br>
     * -u Writes the utilisation for each processor.<br>
     * -h, -?, -help Shows the help file for the system.<br>
     */
    public static void main(String[] args){
```

```
boolean sched = false;
boolean all = false;
boolean trace = false;
boolean runTest = false;
boolean printUtil = false;
boolean explanation = false;
String strSA = "";

byte[] sa;
int round=0;
boolean help=false;

BufferedReader in;
String strLine;
int granularity = 1;
try {
    Verifier verify = new Verifier();
    for (int i = 0; i<args.length; i++) {
        if(args[i].equals("-s")) {
            sched = true;
            verify.allSchedulingAlg = true;
        }
        else if(args[i].equals("-a")) {
            verify.allSystems = true;
            all = true;
        }
        else if (args[i].equals("-g")) {
            granularity = Integer.parseInt(args[++i]);
        }
        else if(args[i].equals("-test")) {
            runTest = true;
        }
        else if(args[i].equals("-e")) {
            verify.explain = true;
        }
        else if (args[i].equals("-f")) {
            verify.queryFile = args[++i];
        }
        else if (args[i].equals("-o")) {
            verify.fl = args[++i];
        }
        else if(args[i].equals("-h")
                || args[i].equals("-help")
                || args[i].equals("-?")) {
            help = true;

            in = new BufferedReader(new FileReader("help.txt"));
            while ((strLine = in.readLine()) != null) {
                System.out.println(strLine);
            }
        }
        else if(args[i].indexOf("-t")>=0) {
            verify.trace = true;
            if (args[i].length()>2) {
                verify.limit = Integer.parseInt(args[i].substring(2));
            }
            else {
                verify.limit = 0;
            }
        }
        else if ((args[i].indexOf("-u"))>=0) {
            verify.printUtil = true;
        }
        else {
            strSA += args[i] + " ";
        }
    }
    if (!help) {
        if (!strSA.equals("")) {
            String[] strSaArr =  strSA .split("\\s");
            sa = new byte[strSaArr.length];
            for (int i = 0; i<sa.length; i++) {
                if (strSaArr[i].equals("EDF"))
                    sa[i] = Processor.EDF;
                else if(strSaArr[i].equals("RM"))
                    sa[i] = Processor.RM;
                else if(strSaArr[i].equals("DM"))
                    sa[i] = Processor.DM;
                else if(strSaArr[i].equals("FP"))
                    sa[i] = Processor.FP;
                else {
                    throw new Exception();
                }
            }
        }
        else {
            sa = new byte[0];
```

```
            }
            verify.sa = sa;

            MPSoC sys1 = new MPSoC(granularity);
            verify.apps = sys1.apps;
            verify.pl = sys1.pl;

            try {
                if (runTest) {
                    boolean test = false;
                    verify.inTestMode = true;
                    verify.trace = false;
                    verify.allSystems = all;
                    for (int i= 2; i<=2; i++) {
                        for (int m = 1; m<=1; m++) {
                            for (int n = 1; n <=200; n++) {
                                if (i == 0 && m<65) {
                                    m = 65;
                                    n = 2;
                                }
                                test = false;
                                try {
                                    System.out.print(m + ";" + n + ";");
                                    verify = new Verifier();
                                    //sys1 = new MPSoC(m,n,i);
                                    verify.inTestMode = true;
                                    verify.apps = sys1.apps;
                                    verify.pl = sys1.pl;

                                    verify.executeSystem();
                                }
                                catch(Exception e)
                                     {System.out.print("Exception\n"); break; }
                            }
                        }
                    }
                }
                else if (sched && sa.length>0) {
                    verify.runAllSchedules();
                }
                else if (all) {
                    verify.assignTaskToProc();
                }
                else {
                    verify.executeSystem();
                }
            }

            catch (Exception e) {
                e.printStackTrace();
                new Error("An error occured during verification.\n" +
                          "This might be due to an out of memory-error " +
                          "from Verifyta, but can also be caused by " +
                          "errors in the constructed UPPAAL model.\n" +
                          "Verify the system in MPSoC.java file and try
                              again\n");
            }
        }
    }

    catch (Exception e) {
        e.printStackTrace();
        new Error("An error occured:\n" +
                  "Provided arguments is not recognized\n" +
                  "use: java MOVES -help to see allowed arguments\n");
    }
  }
}
```

E.2 Processor.java

```
/**
* Processor.java
* Processor class for MoVES.
* The Processor class contains information about the
* Processors created in the MPSoC system designed in MPSoC.java
*
* A Processor is constructed using a speed, scheduling algorithm
* and allocation algorithm.
*
* @author   Kristian Staaloe Knudsen, Jens Ellebaek Nielsen
```

```
* @version 1.1
* @access public
*/
package MoVES;

public class Processor{
    /** The id of the processor */
    public int procID;
    /** The scheduling algorithm for the processor */
    public int schedType;
    /** The frequency for the processor defined in Hz */
    public int frequency;
    /** Static processor count */
    private static int id=0;
    /** The allocation algorithm specified */
    public int allocation;

   /**
    * Constructor.
    * @param speed Integer containing the processor frequency in Hz
    * @param sch Scheduling algorithm defined below.
    * @param allocation integervariable containing information about the allocation
         algorithm.
    */
    public Processor(int speed, byte sch, int allocation){
        procID=++id;
        schedType=sch;
        frequency = (speed);
        this.allocation = allocation;
    }


    /** Preemptive Critical Section allocation protocol */
    public static final int PCS = 0;
    /** Priority Inheritance allocation protocol */
    public static final int PRI_INH = 1;
    /** Non-Preemptive Critical Section allocation protocol */
    public static final int NPCS = 2;

    /** Earliest Deadline First scheduling */
    public static final byte EDF = 1;
    /** Rate Monotonic scheduling */
    public static final byte RM = 2;
    /** Deadline Monotonic scheduling */
    public static final byte DM = 3;
    /** Fixed Priority scheduling */
    public static final byte FP = 4;
}
```

E.3 Task.java

```
/**
* Task.java
* Task class for MoVES.
* The Task class handles the tasks defined in MPSoC.java.
*
* A task can be defined as either periodic or nonperiodic.
* Further different deadline types can be set for the different tasks.
*
* @author Kristian Staaloe Knudsen, Jens Ellebaek Nielsen
* @version 1.1
* @access public
*/
package MoVES;

public class Task{
    /** Maximal executiontime in cycles */
    public int emax;
    /** Minimal executiontime in cycles */
    public int emin;
    /** Deadline in cycles */
    public int d;
    /** Offset in cycles */
    public int o;
    /** Period in cycles */
    public int p;
    /** Fixed Priority for the task */
    public int pri;
    /** The processor the task is mapped onto */
    public int proc=0;
    /** Deadline in seconds */
```

```
    public double dtime;
    /** Period in seconds */
    public int ptime;

    /** Minimum execution time in cycles not according to processor */
    public int emin_org;
    /** Maximum execution time in cycles not according to processor */
    public int emax_org;
    /** Offset in cycles not according to processor */
    public int otime;
    /** The tasks deadline property 0 = Hard, 1 = Soft, 2 = Firm */
    public int sd = 0;
    /** The id of the task */
    public int taskID;
    boolean limitTask = false;
    /** Static counter */
    public static int taskCount=0;

    /** Is the offset exceedet in the verifier */
    public Boolean offsetExceeded = false;
    /** How many periods is to be executed */
    public int maxCount = 0;
    /** Is the task non-periodic */
    public boolean nonperiodic;

   /**
    * Constructor for periodic tasks
    * @param bcet Integer for Best case execution times.
    * @param wcet Integer for West case execution times.
    * @param deadline Defined in seconds.
    * @param offset Defined in seconds.
    * @param period Defined in seconds.
    * @param priority Integer containing the fixed priority
    */
    public Task(int bcet, int wcet, double deadline, int offset, double period, int
         priority){
        dtime=deadline;
        otime=offset;
        ptime=period;
        pri=priority;
        taskID = ++taskCount;
        emin_org = bcet;
        emax_org = wcet;
        nonperiodic = false;
    }

   /**
    * Overloaded constructor for non-periodic tasks
    * @param bcet Integer for Best case execution times.
    * @param wcet Integer for Worst case execution times.
    * @param deadline Defined in seconds.
    * @param priority Integer containing the fixed priority
    */
    public Task(int bcet, int wcet, double deadline, int priority) {
        nonperiodic = true;
        emin_org = bcet;
        emax_org = wcet;
        dtime = deadline;
        pri = priority;
        taskID = ++taskCount;
    }

   /**
    * Set a task to be soft-deadlined
    */
    void softDeadline() {
        sd = 1;
    }

   /**
    * Set a task to be firm-deadlined
    */
    void firmDeadline() {
        sd = 2;
    }
}
```

E.4 Environment.java

```
/**
* Environment.java
```

```
* Environment class for MoVES.
* The Environment class contains information about the
* environments created for triggered tasks in the constructed
* MPSoC system from MPSoC.java
*
* A Environment is constructed using a Task and an minimum interarrival time
*
* @author    Kristian Staaloe Knudsen, Jens Ellebaek Nielsen
* @version   1.1
* @access    public
*/
package MoVES;

public class Environment{
    /** Static counter to ensure uniqueness of the environments */
    private static int count=0;

    /** Environment[] containing all environments in the system */
    public static Environment[] ea = new Environment[0];
    /** The interarrival time for the environment */
    public double interarrival;
    /** The interarrival time in cycles */
    public int interarrival_cycles = 0;

    /** The task id for the task which is to be triggered */
    public int taskid;
    /** Mapped task id */
    public int mapped_task;

   /**
    * Constructor
    * @param task The task which can be triggered by the environment
    * @param interarrival A double containing the minimum interarrivaltime.
    */
    public Environment(Task task, double interarrival){
        ++count;
        taskid = t.taskID;
        this.interarrival = interarrival;
        Environment[] temp = new Environment[count];
        for (int i = 0; i<count-1; i++) {
            temp[i] = ea[i];
        }
        temp[count-1] = this;
        ea = new Environment[count];
        for (int i = 0; i<count; i++)
            ea[i] = temp[i];
    }

   /**
    * An empty constructor, from which the global declarations can be found.
    */
    public Environment()
    {
        //Empty constructor
    }
}
```

E.5 Application.java

```
/**
* Application.java
* Application class for MoVES.
* The Application class contains information about the
* application in the constructed MPSoC system from MPSoC.java
* Application handles:
*   - Tasks
*   - Dependencies between tasks
*   - Resources
*   - Which tasks uses the resources?
*
* The application is constructed using a Task[][] containing a mapping
* of tasks onto processing elements.
* A cost[] which contains the different costs designed for the current system.
* Along with a granularity, which is used later on by BuildTA class to calculate
* the correct execution times.
*
* @author    Kristian Staaloe Knudsen, Jens Ellebaek Nielsen
* @version   1.1
* @access    public
*/
package MoVES;
```

```java
public class Application{
    /** Containing the tasks in the system */
    public Task[][] tasks;
    /** The dependencies between tasks */
    public int[][] dependencies;

    /** Which tasks uses any resource */
    public int[][] resourcesUsed;
    /** An array containing all costs in the system */
    public Cost[] ca;

    /** The dependencies as they are produced (not accoring to mapping) */
    public int[][] origDep;
    /** The granularity specified by the user */
    public int granularity;

   /**
    * Constructor.
    * @param ts A variable of type Task[][]
    * @param ca A Cost[] variable containing information about desired costs.
    * @param granularity A integer variable containing a desired granularity
    */
    public Application(Task[][] ts, Cost[] ca, int granularity){
        tasks=ts;
        this.ca = ca;
        dependencies = new int[nrTasks()][nrTasks()];
        origDep = new int[200][2];

        for(int i=0; i<origDep.length; i++)
            for (int j=0; j<origDep[i].length; j++)
                origDep[i][j] = 0;

        for(int i=0; i<nrTasks(); i++)
            for(int j=0; j<nrTasks(); j++)
                dependencies[i][j]=0;
        setTaskOnProc();
        Resource temp = new Resource();
        if (temp.rs.length-1>0) {
            resourcesUsed = new int[nrTasks()][temp.rs.length-1];
            for (int i = 0; i<resourcesUsed.length; i++)
                for (int j = 0; j<resourcesUsed[i].length; j++)
                    resourcesUsed[i][j] = 0;
        }
        else {
            resourcesUsed = new int[nrTasks()][1];
            for (int i = 0; i<resourcesUsed.length; i++)
                for (int j = 0; j<resourcesUsed[i].length; j++)
                    resourcesUsed[i][j] = 0;
        }
        this.granularity = granularity;
    }


   /**
    * Is there any tasks with soft deadlines?
    * @return A boolean if there is any soft deadlines in the system.
    */
    public boolean softDeadlines() {
        for (int i = 0; i<tasks.length; i++)
            for (int j = 0; j<tasks[i].length; j++)
                if (tasks[i][j].sd == 1)
                    return true;
        return false;
    }

   /**
    * Find the total number of tasks
    * @return The number of tasks as integer
    */
    public int nrTasks(){
        int nr=0;
        for(int i=0;i<tasks.length;i++)
            nr+=tasks[i].length;
        return nr;
    }

   /**
    * Assign task to processor according to double-array
    */
    private void setTaskOnProc() {
        for(int i = 0; i<tasks.length; i++) {
            for (int j = 0; j<tasks[i].length; j++) {
                tasks[i][j].proc = i+1;
            }
        }
    }
```

```
    /**
     * Add dependencies between tasks
     */
    public void addDep(Task from, Task to) {
        for (int i = 0; i<origDep.length; i++) {
            if (origDep[i][0] == 0) {
                origDep[i][0] = from.taskID;
                origDep[i][1] = to.taskID;
                break;
            }
        }
    }

    /**
     * Set a task to use a resource
     */
    public void useResource(Task t, Resource r) {
        resourcesUsed[t.taskID-1][r.id-1] = 1;
    }

    /**
     * Update dependencies (if tasks array has changed)
     */
    public void updateDependency() {
        int s = 1;
        int newFrom=0, newTo=0;
        dependencies = new int[nrTasks()][nrTasks()];
        for(int i=0; i<nrTasks(); i++)
            for(int j=0; j<nrTasks(); j++)
                dependencies[i][j]=0;

        for (int n = 0; n < origDep.length; n++) {
            s = 1;
            for (int m=0; m<tasks.length; m++) {
                for (int l = 0; l<tasks[m].length; l++) {

                    if(tasks[m][l].taskID == origDep[n][0]) {
                        newFrom = s;
                    }
                    if (tasks[m][l].taskID == origDep[n][1]) {
                        newTo = s;
                    }
                    s++;
                }
            }
            if (newFrom>0 && newTo>0) {
                dependencies[newTo-1][newFrom-1]=1;
            }
        }
    }
}
```

E.6 Verifier.java

```
/**
* Verifier.java
* Verifier class for MoVES.
* The Verifier class handles the verification of MPSoCs defined in MPSoC.java.
*
* The Verifier uses Verifyta to handle verification. By using BuildTA and Parser
* the Verifier is able to generate a XML-file and verify this. And finally create a
    useful output to the user..
*
* @author    Kristian Staaloe Knudsen, Jens Ellebaek Nielsen
* @version   1.1
* @access    public
*/
package MoVES;

import java.io.*;

public class Verifier {
    /** The name of the file, in which the UPPAAL model is stored */
    public String fl = "systemXML";
    /** The file containing the queries which are verified (default: bool.q)*/
    public String queryFile = "bool.q";

    /** How many times should the system execute?*/
    public int intRounds = 1;
    /** The desired limit for the execution */
```

```
public int limit =0;

/** Contains the hyperperiod of the system */
public int hyperPeriod = 0;

/** Contains the scheduling algorithms which is provided by the user */
public byte[] sa;

/** Counter */
private static int number = 0;

/** The application for the MPSoC system */
public Application apps;
/** The platform for the MPSoC system */
public Platform pl;

/** Boolean to see if the system is in test-mode */
public boolean inTestMode = false;
/** Boolean tells if a schedule is wanted */
public boolean trace = false;
/** Should the tasks be mapped onto all processors? */
public boolean allSystems = false;
/** Should the system be testet with several schedulig algorithms */
public boolean allSchedulingAlg = false;
/** Print the utilisation during verification */
public boolean printUtil = false;
/** Print an explanation of the trace after verification */
public boolean explain = false;


/**
 * Empty constructor
 * @exception Exception if the verifier runs out of memory
 */
public Verifier() throws Exception {
    //Empty constructor
}

/**
 * Sets the max number of rounds for a task according to the limit
 * specified by the user. If no limit is specified, the hyperperiod is used.
 * @exception Exception if the verifier runs out of memory
 */
private void setMaxRounds() throws Exception{
    Task[] ta = toSingleArray(apps.tasks);
    if (apps.softDeadlines()){
        for (int i = 0; i<ta.length; i++) {
            if (!ta[i].nonperiodic)
                ta[i].maxCount = 0;
        }
    }
    else {
        hyperPeriod = pl.calcHyperPeriod(ta);
        hyperPeriod = hyperPeriod*2;
        if (inTestMode)
            System.out.print(hyperPeriod + ";");
        for (int i =0; i<ta.length; i++) {
            if (!ta[i].nonperiodic) {
                if (limit == 0) {
                    ta[i].maxCount =
                         (int)Math.ceil(hyperPeriod/(double)ta[i].p);
                }
                else {
                    ta[i].maxCount = (int)Math.ceil(limit/(double)ta[i].p);
                }
            }
        }
    }
}

/**
 * Can run all schedules, meaning each processor uses the scheduling
 * algorithm given as argument for MoVES
 * @exception Exception if the verifier runs out of memory
 */
public void runAllSchedules() throws Exception{
    auxRunAllSchedules(0, pl.processors);
}

/**
 * Aux to run all schedules.
 * @param p An iterator
 * @param ps The processor array
 * @exception Exception if the verifier runs out of memory
 */
public void auxRunAllSchedules(int p, Processor[] ps) throws Exception {
    if (p==ps.length) {
```

```
            if (allSystems) {
                auxAssignTaskToProc(0, toSingleArray(apps.tasks), ps);
            }
            else {
                executeSystem();
            }
        }
        else {
            for (int i = 0; i<sa.length; i++) {
                ps[p].schedType = sa[i];
                auxRunAllSchedules(p+1, ps);
            }
        }
    }

    /**
     * Sets tasks to processor using the auxAssignTaskToProc-function.
     */
    public void assignTaskToProc() throws Exception {
        auxAssignTaskToProc(0, toSingleArray(apps.tasks), pl.processors);
    }

    /**
     * Aux to assign task to processor.
     * @param p Iterator
     * @param ta Task[] containing all tasks in the system.
     * @param ps Processor[] containing all processors of the system.
     * @exception Exception if the verifier runs out of memory
     */
    public void auxAssignTaskToProc(int p, Task[] ta, Processor[] ps) throws
         Exception {
        if (p==ta.length) {
            toDoubleArray(ta, ps);
            executeSystem();
        }
        else {
            for (int i = 1; i<=ps.length; i++) {
                ta[p].proc = i;
                auxAssignTaskToProc(p+1, ta, ps);
            }
        }
    }

    /**
     * Generates a single[] of tasks.
     * @param ts Task[][] containing all tasks
     * @return Task[] containing all tasks.
     */
    public Task[] toSingleArray(Task[][] ts) {
        int nr=0;
        for(int i=0;i<ts.length;i++) {
            nr+=ts[i].length;
        }
        Task[] taskArr = new Task[nr];
        nr = 0;
        for (int i = 0; i<ts.length; i++) {
            for (int j=0; j<ts[i].length; j++) {
                taskArr[nr] = ts[i][j];
                nr++;
            }
        }
        return taskArr;
    }

    /**
     * Converts a single-array to a double[][].
     * @param ta Task[] which is to be converted
     * @param ps Processor[] containing all processors
     */
    public void toDoubleArray(Task[] ta, Processor[] ps) {
        Task[] temp = {};
        int[] intTemp = new int[ps.length];
        for (int i = 0; i<ps.length; i++) {
            intTemp[i] = 0;
            for(int j=0; j<ta.length; j++) {
                if (ta[j].proc == ps[i].procID) {
                    intTemp[i]++;
                }
            }
        }

        int iterator = 0;

        Task[][] temp_ta = new Task[ps.length][];
        for (int j=0;j<ps.length;j++) {
            temp_ta[j] = new Task[intTemp[j]];
            iterator = 0;
```

```
        for(int i =0;i<ta.length;i++) {
            if (ta[i].proc == ps[j].procID) {
                temp_ta[j][iterator] = ta[i];
                iterator++;
            }
        }
    }
    apps.tasks = temp_ta;
}

/**
 * Generates an XML-system using the BuildTA-class.
 * Verifies the system with Verifyta
 * Produces an output to the user according to the provided arguments
 * @exception Exception If Verifyta runs out of memory.
 * @return Boolean containing info about result of verification.
 */
public boolean executeSystem() throws Exception{
    long startTime;
    long totalTime = 0;
    long execTime = 0;

    String strLine = "";
    String[][] result = new String[10][2];
    boolean boolSatis = false;
    boolean lineFoundInFile=false;


    // Read a line of text
    BufferedReader bur, inErr;
    BufferedInputStream bis;
    DataInputStream in;

    number++;

    totalTime = 0;
    apps.updateDependency();
    //pl.assignProcessorSpeed(apps);

    for (int i = 0; i<result.length; i++)
        result[i][0] = "";
    String str = "";
    try {
        bur = new BufferedReader(new FileReader(queryFile));
    }
    catch (FileNotFoundException fnfe) {
        System.out.println("Queryfile not found. Please locate the file and try
            again");
        return false;
    }
    int i = 0;
    while ((str = bur.readLine()) != null) {
        if (str.indexOf("E<>") == 0
            ||str.indexOf("A<>") == 0
            ||str.indexOf("A[]") == 0
            ||str.indexOf("E[]") == 0) {
            result[i++][0] = str;
        }
    }

    PrintStream fout = new PrintStream(new File(fl+".xml"));

    BuildTA translator = new BuildTA(apps,pl, fout);

    setMaxRounds();

    Parser parse = new Parser(toSingleArray(apps.tasks));

    pl.calcUtilisation(toSingleArray(apps.tasks));

    if (allSchedulingAlg) {
        parse.printSched(pl.processors);
    }
    if (printUtil) {
        pl.printUtilisation();
    }

    if (!pl.checkUtil() && !trace) {
        System.out.print("Utilisation higher than 1.00\nSystem not
            schedulable\n");
        return false;
    }
    else {
        if(apps.softDeadlines()) {
            for (i = 0; i<result.length; i++) {
                if (result[i][0].indexOf("allFinish()")>0) {
```

```
                System.out.println("Cannot verify allFinish() property when
                    using soft-deadlines");
                return false;
            }
        }

    }
    if (!inTestMode && explain)
        System.out.println("Generating XML-system");
    translator.mkXML();
    if (!inTestMode && explain)
        System.out.println("Verifying");

    if (allSystems) {
        parse.printTasks(toSingleArray(apps.tasks));
    }

    Process p = null;
    Runtime rt;
    rt = Runtime.getRuntime();

    for (i = 1; i<=intRounds; i++) {
        startTime = System.currentTimeMillis();
        if (inTestMode) {
            //p=rt.exec("cmd /c verifyta -o 1 -S 2 -t 0 -y " + fl + ".xml
                bool.q > temp/temp" +number +  ".txt");
            p=rt.exec("cmd /c verifyta -o 1 " + fl + ".xml " + queryFile +
                " > temp/temp" +number +  ".txt");
        }
        else if (trace) {
            p=rt.exec("cmd /c verifyta -t 0 -S 2 -y "+ fl + ".xml " +
                queryFile + " > temp.txt");
        }
        else {
            p=rt.exec("cmd /c verifyta -T -S 2 "+ fl + ".xml " + queryFile +
                " > temp.txt");
        }

        if (inTestMode || !trace) {
            p.waitFor();
        }
        inErr = new BufferedReader(new
            InputStreamReader(p.getErrorStream()));

        if (!inTestMode && trace) {
            while ((strLine = inErr.readLine()) != null) {
                if (strLine.indexOf("(")==0) {
                    parse.parseTrace(strLine);
                }
            }
        }

        execTime = (System.currentTimeMillis()-startTime);

        if (inTestMode)
            System.out.print(execTime + ";");

        // Read a line of text
        if (inTestMode) {
            bur = new BufferedReader(new FileReader("temp/temp" + number +
                ".txt"));
        }
        else {
            bur = new BufferedReader(new FileReader("temp.txt"));
        }
        i = 0;
        while ((strLine = bur.readLine()) != null)   {
            if (strLine.indexOf("Property is NOT satisfied.") > 0) {
                boolSatis = false;
                result[i++][1] = "false";
                lineFoundInFile = true;
            }
            if (strLine.indexOf("Property is satisfied.") > 0) {
                boolSatis = true;
                result[i++][1] = "true";
                lineFoundInFile = true;
            }
            if (strLine.indexOf("Out of memory") >= 0) {
                //System.out.println("excep");
                if (inTestMode) {
                    throw new Exception();
                }
                result[i++][1] = "Out of memory";
                lineFoundInFile = true;
            }
        }
    }
```

```
            if (lineFoundInFile) {
                if (inTestMode)
                    System.out.print(boolSatis ? "True;\n":";False;\n");
                else {
                    System.out.print("Verification time: ");
                    execTime = execTime/1000;
                    int rest = 0;
                    if (execTime > 60) {
                        System.out.print(execTime/60);
                    }
                    else {
                        System.out.print(0);
                    }
                    System.out.print(".");
                    if (execTime%60>=10) {
                        System.out.print((execTime%60));
                    }
                    else {
                        System.out.print("0" + (execTime%60));
                    }
                    System.out.print(" min\n");

                    for (i = 0; i<result.length; i++) {
                        if (!result[i][0].equals(""))
                            System.out.println(result[i][0] + ": " + result[i][1]);
                    }
                }
            }
            else {
                throw new Exception();
            }

            if (!inTestMode && trace) {
                parse.printTrace();
                if (explain) {
                    parse.printExplain();
                }
            }
        }
        return boolSatis;
    }
}
```

E.7 BuildTA.java

```
/**
* BuildTA.java
* BuildTA class for MoVES.
* The BuildTA class generates the timed-automata model for UPPAAL
* The model is constructed in a XML-file and stored using the printstream fout.
*
* The BuildTA class calculates the execution cycles for each task, and assigns these
* to the tasks before generating the model.
*
* The BuildTA class uses an Application, Platform and a printstream to generate the
    model.
*
* @author    Kristian Staaloe Knudsen, Jens Ellebaek Nielsen
* @version   1.1
* @access    public
*/
package MoVES;

import java.io.*;

public class BuildTA{
    /** The application for the MPSoC system */
    private static Application application;

    /** The platform for the MPSoC system */
    private static Platform platform;

    /** The printStream which writes the XML-file */
    public static PrintStream fout;
    /** All environments specified in the system */
    public static Environment[] environments;

   /**
    * Constructor
    * @param app Containing the application
```

```java
 * @param pl containing the platform
 * @param fout of type PrintStream
 */
public BuildTA(Application app, Platform pl, PrintStream fout){
    application = app;
    platform = pl;
    this.fout = fout;

    int g;

    Environment eTemp = new Environment();
    environments = new Environment[eTemp.ea.length];
    if (eTemp.ea.length > 0) {
        for (int i = 0; i< eTemp.ea.length; i++) {
            g = 1;
            for (int j = 0; j< application.tasks.length; j++) {
                for (int k = 0; k<application.tasks[j].length; k++) {
                    if (application.tasks[j][k].taskID == eTemp.ea[i].taskid) {
                        eTemp.ea[i].mapped_task = g;
                    }
                    else
                        g++;
                }
            }
            environments[i] = eTemp.ea[i];
        }
    }


    setPeriods();
}

/**
 * Calculates periods for tasks according to the speed of their execution
 * platforms. The periods is in the end divided by the granularity, and rounded
 * up.
 */
private static void setPeriods() {
    int lcm = platform.calcLCM(platform.procSpeed(), 0, 1);
    for (int i = 0; i<platform.processors.length;i++) {
        for (int j = 0; j<application.tasks[i].length; j++) {
            if (!application.tasks[i][j].nonperiodic) {
                application.tasks[i][j].p =
                    (int)((lcm*application.tasks[i][j].ptime)/
                    application.granularity);
            }
            application.tasks[i][j].d =
                (int)((lcm*application.tasks[i][j].dtime)/
                application.granularity);
            application.tasks[i][j].emax =
                (int)Math.ceil(application.tasks[i][j].emax_org*
                (double)(lcm/platform.processors[application.tasks[i][j].proc-1].frequency)/
                 (double)application.granularity);
            application.tasks[i][j].emin =
                (int)Math.ceil(application.tasks[i][j].emin_org*
                (double)(lcm/platform.processors[application.tasks[i][j].proc-1].frequency)/
                 (double)application.granularity);
            application.tasks[i][j].o =
                (int)((lcm*application.tasks[i][j].otime)/
                (double)application.granularity);

        }
    }
    for (int i = 0; i<environments.length;i++) {
      environments[i].interarrival_cycles = (int)
           (lcm*environments[i].interarrival/application.granularity);
    }
}

/**
 * Return the application
 * @return Application
 */
public Application getApp() {
    return application;
}

/**
 * Converts the dependency[][] to a string used in the model.
 * @param int[][] containing dependencies.
 * Written to fout
 */
private static void toStringDep(int[][] d){
    fout.print("{{");
    for (int i=0;i<d.length;i++){
        for (int j=0;j<d[0].length;j++){
            if(j!=(d[0].length-1))
```

```
                    fout.print(d[i][j]+",");
                else
                    fout.print(d[i][j]);
            }
            if(i==(d.length-1)) {
                fout.print("}};\n");
            }
            else {
                fout.print("},{");
            }
        }
    }

   /**
    * Converts the specified environments to a string, which is used in the model.
    * the string is written directly in fout.
    */
    private static void toStringEnvironment() {
        String en = "";
        for (int i = 0; i<environments.length;i++) {
            en += "env" + i + " = Environment("+environments[i].mapped_task+",
                 trigger["+(environments[i].mapped_task-1)+"], " +
                 environments[i].interarrival_cycles+");\n";
        }
        fout.println(en);
    }

   /**
    * Tasks is converted to a string.
    * @param tas The task which is to be converted.
    * @param l A local id of type int.
    * @param g A global id of type int.
    * The result is written on fout.
    */
    private static void toStringTasks(Task tas,int l, int g){
        String task;
        String deadlineType = "";
        if (tas.sd == 1)
            deadlineType = "SOFT";
        else if(tas.sd == 2)
            deadlineType = "FIRM";
        else
            deadlineType = "HARD";

        if (!tas.nonperiodic) {
            task = "Taskc"+(g+1)+" = Task("+(tas.proc)+", "+(g+1)+", "+(l+1)+",
                 "+tas.emin+", "+tas.emax+", "+tas.d+", "+tas.o+", "+tas.p+",
                 "+tas.pri+", tid["+((tas.proc)-1)+"], gtid["+((tas.proc)-1)+"],
                 sact["+((tas.proc)-1)+"][REA], sact["+((tas.proc)-1)+"][RUN],
                 sact["+((tas.proc)-1)+"][PRE], sact["+((tas.proc)-1)+"][FIN],
                 "+deadlineType+", " + tas.maxCount + ");";
        }
        else {
            task = "Taskc"+(g+1)+" = Triggered("+(tas.proc)+", "+(g+1)+", "+(l+1)+",
                  "+tas.emin+", "+tas.emax+", "+tas.d+", "+tas.pri+",
                 tid["+((tas.proc)-1)+"], gtid["+((tas.proc)-1)+"],
                 trigger["+(g)+"], sact["+((tas.proc)-1)+"][REA],
                 sact["+((tas.proc)-1)+"][RUN], sact["+((tas.proc)-1)+"][PRE],
                 sact["+((tas.proc)-1)+"][FIN], "+deadlineType+");";
        }
        fout.println(task);
    }

   /**
    * Finds the os for each processing element and writes
    * them as an array to fout.
    */
    private static void typeOfProc() {
      String test = "int[0,3] processorScheduling[M] = {";
      for (int i = 0; i<(platform.processors).length; i++) {
        if (i > 0) {
          test += ", ";
        }
        test += typeProc(platform.processors[i]);
      }
      test += "}; \n//Contains information about the scheduling principle for each
           processor.\n";
      fout.println(test);
    }

   /**
    * Get the type of allocation algorithm
    * @param pr The processor to find algorithm for
    * @return String containing allocation algorithm.
    */
    private static String typeAll(Processor pr) {
        switch(pr.allocation) {
```

```
        case Processor.PCS:
            return "PCS";
        case Processor.PRI_INH:
            return "PRI_INH";
        case Processor.NPCS:
            return "NPCS";
        }
        return "";
    }

    /**
     * Get the type of scheduling algorithm
     * @param pr The processor to find algorithm for
     * @return String containing scheduling algorithm.
     */
    private static String typeProc(Processor pr){
        switch(pr.schedType){
        case Processor.EDF:
            return "EDF";

        case Processor.RM:
            return "RM";

        case Processor.DM:
            return "DM";

        case Processor.FP:
            return "FP";
        }
        return "";
    }

    /**
     * Returns the number of resources in the system.
     * @return int The number of resources.
     */
    private static int nrOfResources() {
      return application.resourcesUsed[0].length;
    }

    /**
     * Converts the needed resources array into a string which is
     * written directly to fout.
     */
    private static void toStringNeeded() {
      String res = "bool NeededResources[M][N][R] = ";
      for (int j = 0; j<nrOfProc(platform.processors);j++) {
        if (j>0)
          res += ",";
        if (j == 0)
          res += "{";
        for (int i=0;i<maxNrTasks(application.tasks);i++) {
          if (i > 0)
            res += ",";
          if (i == 0) {
            res += "{";
          }
          for (int k = 0; k<nrOfResources(); k++) {
            if (k == 0) {
              res += "{";
            }
            else {
              res += ",";
            }
            try {
                res +=
                     application.resourcesUsed[application.tasks[j][i].taskID-1][k];
            }
            catch (ArrayIndexOutOfBoundsException aioobe) { res+= "0"; }
            if (k == nrOfResources()-1)
              res += "}";
          }
          if (i==maxNrTasks(application.tasks)-1) {
            fout.print(res + "}");
            res = "";
          }
        }
        if (j ==nrOfProc(platform.processors)-1)
        fout.print("};\n");
      }
    }


    /**
     * The processing element (controller, synchronizer, allocation and
     * scheduler is described as strings. These strings are written on fout.
     */
```

```
 private static void toStringProc(Processor pr, int i){
      fout.println("Con"+(i+1)+" = Control("+(i+1)+", ftid["+i+"], tid["+i+"],
           gtid["+i+"], cact["+i+"][SYN], cact["+i+"][SCH], cact["+i+"][ALL],
           sact["+i+"][REA], sact["+i+"][FIN], sact["+i+"][RUN],
           sact["+i+"][PRE]);\n"+
 "Syn"+(i+1)+" = Synchronizer("+(i+1)+", ftid["+i+"], cact["+i+"][SYN]);\n"+
 "Sch"+(i+1)+" = Scheduler("+(i+1)+", ftid["+i+"], cact["+i+"][SCH]);\n"+
 "Alloc"+(i+1)+" = Allocator("+(i+1)+", " + typeAll(pr) + ",
       cact["+i+"][ALL]);");
 }

/**
 * Generates the system specification in UPPAAL.
 * uses the functions: toStringProc and ToStringTasks.
 * @param ta Task[][] containing the task mapping array
 * @param pa Processor[] containing the processors of the system.
 */
 private static void toStringAll(Task[][] ta, Processor[] pa){
      String rs = "";
      int gbl=0;
      for (int i=0; i<ta.length; i++){
           if (ta[i]!=null){
                toStringProc(pa[i],i);
                for (int j=0; j<ta[i].length; j++){
                     if (ta[i][j]!=null){
                          toStringTasks(ta[i][j],j,gbl);
                          gbl++;
                     }
                }
           }
      }
      toStringEnvironment();
 }

/**
 * Returns the total number of processors.
 * @return int The total number of processors.
 */
 private static int nrOfProc(Processor[] p) {return p.length;}

/**
 * Returns the maximum number of tasks on one processor.
 * @return int The maximum number of tasks on one processor.
 */
 private static int maxNrTasks(Task[][] t) {
      int mx = 0;
      for(int i=0;i<t.length;i++)
           if(t[i].length>mx)
                mx=t[i].length;
      return mx;
 }

/**
 * Returns the total number of tasks in the system.
 * @return int The total number of tasks in the system.
 */
 private static int totalNrTasks(Task[][] t){
      int cnt=0;
      for (int i=0; i<t.length; i++){
           if (t[i]!=null){
                for(int j=0; j<t[i].length; j++){
                     if (t[i][j]!=null)
                          cnt++;
                }
           }
      }
      return cnt;
 }

/**
 * Creates the localToGlobal integer[][] in the UPPAAL model
 * containing information about which global-ids is stored on which
 * processor. Written to fout
 * @param t A task[][] with all tasks.
 */
 private static void localToGlobal(Task[][] t){
      int coun = 1;
      String res = "int[0,MN] ltog[M][N]={{";
      for(int i=0; i<t.length;i++){
           for(int j=0; j<maxNrTasks(t);j++){
                if (j<t[i].length){
                     res+=coun;
                     coun++;
                }
                else res+="0";
                if(j!=maxNrTasks(t)-1)
                     res+=", ";
```

```
            }
            res+="}";
            if(i!=t.length-1)
                res+=", {";
        }
        res+="};//global taskids from locals\n";
        fout.println(res);
    }

    /**
     * Creates the system-declaration. Assigns all tasks and processors to the UPPAAL
     * system. Written to fout.
     * @param ta Task[][] containing all tasks
     * @param pa Processor[] containing all processors.
     */
    private static void toStringSystem(Task[][] ta, Processor[] pa){
        String res = "system ";
        String conn = "";
        String syn = "";
        for(int i=1;i<=totalNrTasks(ta);i++){
            if (i<totalNrTasks(ta)) {
                res+="Taskc"+i+ " &lt; "; //", ";
            }
            else {
                res+="Taskc"+i+", ";
            }
        }
        for(int i=1;i<nrOfProc(pa);i++){
            conn+="Con"+i+ " &lt; "; //", ";//" &lt; ";
            syn+="Syn"+i+", Sch"+i+", Alloc"+i+", ";
        }
        conn += "Con"+nrOfProc(pa);
        syn += " Syn"+nrOfProc(pa) + ", Sch" +nrOfProc(pa) + ", Alloc"
            +nrOfProc(pa);
        res += conn + ",";
        res += syn + ", Rescheduler";
        for (int i = 0; i<environments.length; i++) {
            res += ", env" + i;
        }
        fout.print(res);
    }

    /**
     * What is the total number of costs in the system?
     * @return int The total number of costs.
     */
    private static int toStringNrOfCosts() {
        return application.ca.length;
    }

    /**
     * Generates a array for UPPAAL containing information about which tasks
     * uses which costs, and how much of these they are using.
     * Written to the model using fout.
     */
    private static void toStringCostsUsed() {
        fout.print("int totalCost[M][NOC] = {");
        for (int j = 0; j<platform.processors.length;j++) {
            if (j>0)
                fout.print(",");
            fout.print("{");
            for (int i = 0;i<application.ca.length;i++) {
                if (i>0)
                    fout.print(",");
                fout.print("0");
            }
            fout.print ("}");

        }
        fout.print("};\n");

    }

    /**
     * The cost-array is generated.
     */
    private static void toStringCosts() {
        String str = "int costs[MN][NOC][5] = ";
        for (int i = 0; i<totalNrTasks(application.tasks);i++) {
            if (i==0)
                str += "{";
            else
                str += ",";
            for (int j = 0; j<application.ca.length;j++) {
                if (j ==0)
                    str += "{";
                else
```

```
                    str += ",";
                for (int k = 0; k<application.ca[j].costs[i].length;k++) {
                    if (k == 0)
                        str += "{";
                    else
                        str += ",";
                    str += application.ca[j].costs[i][k];
                    if (k == application.ca[j].costs[i].length-1)
                        str += "}";
                }
                if (j == application.ca.length-1) {
                    fout.print(str + "}");
                    str = "";
                }
            }
            if (i==totalNrTasks(application.tasks)-1) {
                fout.print("};\n");
            }
        }
    }

    /**
     * Generates information about shared costs (e.g. memory sharing)
     */
    private static void costsDepend() {
        String str = "int costDepend[MN][MN] = ";
        for (int i = 0; i<application.ca[0].shared.length;i++) {
            if (i==0)
                str += "{";
            else
                str += ",";
            str += "{";
            for (int j = 0; j<application.ca[0].shared[i].length;j++) {
                if (j>0)
                    str += ",";
                str += application.ca[0].shared[i][j];
            }
            fout.print(str+ "}");
            str = "";
            if (i==application.ca[0].shared.length-1)
                fout.print("};\n");
        }
    }

    /**
     * Generates an array of size MNxMN containing zeros to handle
     * the shared costs from the model.
     */
    private static void sharedCosts() {
        String str = "int sharedCost[MN][MN] = ";
        for (int i = 0; i<totalNrTasks(application.tasks);i++) {
            if (i==0)
                str += "{";
            else
                str += ",";
            str += "{";
            for (int j = 0; j<totalNrTasks(application.tasks);j++) {
                if (j>0)
                    str += ",";
                str += "0";
            }
            fout.print(str+ "}");
            str = "";
            if (i==totalNrTasks(application.tasks)-1)
                fout.print("};\n");
        }

    }

    /**
     * Generates the entire XML-system. Makes use of the following functions:
     * nrOfProc, maxNrOfTasks, totalNrOfTasks, nrOfResources, ToStringNrOfCosts,
     * typeOfProc, localToGlobal, toStringUsedCosts, toStringNeeded, sharedCosts,
     * costDepend, toStringAll, toStringSystem
     */
    static void mkXML() {
        fout.print(
"<?xml version='1.0' encoding='utf-8'?><!DOCTYPE nta PUBLIC '-//Uppaal Team//DTD
      Flat System 1.1//EN'
      'http://www.it.uu.se/research/group/darts/uppaal/flat-1_1.dtd'><nta><declaration>\n"
      +
"const int M = "+nrOfProc(platform.processors)+"; //The number of Processors\n" +
"const int N = "+maxNrTasks(application.tasks)+"; //The maximum number of tasks per
      Processor\n" +
"const int MN = "+totalNrTasks(application.tasks)+"; //The total number of tasks\n"
      +
"const int R = " + nrOfResources() + ";//The total number of resources\n" +
```

```
"const int NOC = "+toStringNrOfCosts()+"; //Total number of costs\n" +
"\n" +
"//symbolic representation of scheduling actions\n" +
"const int REA = 0, RUN = 1, PRE = 2, FIN = 3; \n" +
"\n" +
"//symbolic representation of internal processor synchronizing\n" +
"const int SYN = 0, SCH = 1, ALL = 2;\n" +
"\n" +
"//symbolic representation of states for cost-model\n" +
"const int STATIC = 0, IDLE = 1, READY = 2, RUNNING = 3, PREEMPTED=4;\n" +
"//Symbolic representation of memory and power.\n" +
"const int Memory = 0, Power = 1;\n" +
"\n" +
"//Symbolic representation of cost types.\n" +
"const int HARD = 0, SOFT = 1, FIRM = 2;\n" +
"\n" +
"//symbolic representation of scheduling principles\n" +
"const int FP = 0, RM = 1, DM = 2, EDF = 3;\n" +
"\n" +
"//symbolic representation of allocation algorithms\n" +
"const int PCS=0, PRI_INH = 1, NPCS = 2;\n" +
"\n");

typeOfProc();
localToGlobal(application.tasks);

fout.print("broadcast chan reschedule; //broadcast channel for reschelduling after a
      task has finished\n" +
"chan trigger[MN];\n" +
"\n" +
"chan activateReschedule;\n" +
"chan cact[M][3]; //channel array for internal processor actions\n" +
"chan sact[M][4]; //channel array for scheduling actions\n" +
"bool preempted; //set when any task is preempted\n" +
"\n" +
"int[0,N] tid[M]; //transfer of local taskid from task to controller\n" +
"int[0,N] curtid[M]; //variable used to hold the id of the task currently chosen\n"
     +
"int[0,MN] gtid[M]; //transfer of global taskid from task to controller\n" +
"int[0,MN] ftid[M]; //taskid of task which has finished, ftid=0 means no finished
     task\n" +
"int[0,MN] ltid[M]; //variable used to hold the id of the task currently running\n"
     +
"bool Ready[M][N]; //array of tasks which have issued ready signals\n" +
"bool Synchronized[MN]; //array of tasks which are not awaiting dependencies to be
     resolved\n" +
"bool Allocated[MN]; //array of tasks which are not awaiting resources\n" +
"bool taskReadyForTriggering[MN]; //array holding tasks ready for triggering\n" +
"bool preemptedOrIdle[MN]; //Array used in the cost model to test if a task is
     preempted or idle\n" +
"//Array containing the needed resources (as booleans)\n");
toStringNeeded();

fout.print("\n" +
"int UsedResources[M][R]; //array indicating whether a resource is taken\n" +
"\n" +
"bool WaitDep[MN]; //array for tasks which are awaiting dependencies to be
     resolved\n" +
"\n" +
"bool running[M]; //indicating wether a task is currently running on the
     processor\n" +
"\n" +
"bool missedDeadline = false; //Set if a hard-deadline task misses a deadline.\n" +
"bool endState[MN]; //Set when a task reaches its end state.\n" +
"\n");
toStringCostsUsed();

fout.print("\n" +
             "//[task1..taskMN][mem, power][static, idle, ready, run, preempt]\n");
toStringCosts();

fout.print("\n" +
             "//Writer/Reader\n");
costsDepend();

fout.println("\n");
sharedCosts();

fout.println("\n" +
             "//Criteria usable in scheduling\n" +
             "int staticCriteria[M][N]; //criterion used for static scheduling and
                  contains the original parameters for dynamic scheduling.\n" +
             "int dynamicCriteria[M][N]; //criterion used for dynamic scheduling\n"
                  +
             "\n" +
             "//array for original dependencies, 1 for dependency, 0 for no
                  dependency\n" +
```

```
            "\n" +
            "bool origdep[MN][MN]=");
toStringDep(application.dependencies);

fout.print("\n" +
          "//dynamically updated array for current dependencies\n" +
          "bool depend[MN][MN]=");
toStringDep(application.dependencies);
fout.print("\n" +
          "//Locking mechanisms\n" +
          "bool nowrun[M]; //ensuring completion of 'runs' before reacting on
               ready\n" +
          "bool pending[MN]; //ensuring global wait for finish &amp; ready before
               next 'run'\n" +
          "bool h_edf[MN]; //ensuring synchronization on extra state in ready for
               edf\n" +
          "bool h_t[M]; //ensuring completion of all 'runs' before next 'run'\n" +
          "bool h_fin[M]; //ensuring reaction on finish before ready\n" +
          "bool l_in[M]; //ensuring completion of local scheduling before glocal
               reschedule\n" +
          "bool h_r; //ensuring reschedule before next 'run'\n" +
          "bool controllerLock[M]; //External controller lock\n" +
          "bool processing[M]; //Internal controller lock\n" +
          "\n" +
"//function checking for dependencies for task t\n" +
"bool taskHasDependency(int t) {\n" +
"  for (ini : int[0,MN-1]) {    \n" +
"    if (depend[t][ini]) {\n" +
"      return true;\n" +
"    }\n" +
"  }\n" +
"  return false;\n" +
"}\n" +
"\n" +
"//function updating dependencies when task t has finished\n" +
"void opdDep(int t) {\n" +
"  for (ini : int[0,MN-1]) {\n" +
"    depend[ini][t]=false;\n" +
"  }\n" +
"}\n" +
"\n" +
"//function setting the original dependency values for task t\n" +
"void setOrigDep(int t) {\n" +
"  for (ini : int[0,MN-1]) {\n" +
"    depend[t][ini]=origdep[t][ini];\n" +
"  }\n" +
"}\n" +
"\n" +
"//function checking for existance of boolean value in array of size M\n" +
"bool locked(bool la[M]) {\n" +
"  for (ini : int[0,M-1]) {    \n" +
"    if (la[ini]) {\n" +
"      return true;\n" +
"    }\n" +
"  }\n" +
"  return false;\n" +
"}\n" +
"\n" +
"//function checking for existance of boolean value in array of size N\n" +
"bool pend(bool pen[N]) {\n" +
"  bool b = false;\n" +
"  for (ini : int[0,N-1]) {\n" +
"    if (pen[ini] == true) {\n" +
"      return true;  \n" +
"    }\n" +
"  }\n" +
"  return false;\n" +
"}\n" +
"\n" +
"//Function to verify if all tasks has reached their end states\n" +
"bool allFinish() {\n" +
"  for (ini : int[0, MN-1]) {\n" +
"    if (!endState[ini]) {\n" +
"      return false;\n" +
"    }\n" +
"  }\n" +
"  return true;\n" +
"}\n" +
"\n" +
"//function checking for existance of boolean value in array of size MN\n" +
"bool pendMN(bool pen[MN]) {\n" +
"  bool b = false;\n" +
"  for (ini : int[0,MN-1]) {\n" +
"    if (pen[ini] == true) {\n" +
"      return true;  \n" +
"    }\n" +
"  }\n" +
```

```
"  return false;\n" +
"}\n" +
"\n" +
"//Function used to check the total cost in any state\n" +
"int totalCostUsed(int costnr) {\n" +
"  int total = 0;\n" +
"  for (ini : int[0,M-1]) {\n" +
"    total += totalCost[ini][costnr]; \n" +
"  }\n"+
"  return total;\n" +
"}\n" +
"\n" +
"//Function used to see if any ready or finish signal still needs to be recieved\n"
    +
"bool notSignalsPending() {\n" +
"  return (!locked(h_t)\n" +
"          &amp;&amp; !locked(h_fin)\n" +
"          &amp;&amp; !locked(processing)\n" +
"          &amp;&amp; !pendMN(pending)\n" +
"          &amp;&amp; !pendMN(h_edf));\n" +
"}\n" +
"\n" +
"//Function that adds cost to the cost used in beginning of a timeunit.\n" +
"void updateCost(int gtasknr, int schnr, int State) {\n" +
"  if (State == READY &amp;&amp; preemptedOrIdle[gtasknr-1]) {\n" +
"    State = PREEMPTED;\n" +
"  }\n" +
"  for (i :int[0,NOC-1]) {\n" +
"    if (i==Memory &amp;&amp; State == RUNNING) {\n" +
"      int sharedCostTemp = 0;\n" +
"      for (j: int[0, MN-1]) {\n" +
"        sharedCostTemp += sharedCost[gtasknr-1][j];\n" +
"        sharedCost[gtasknr-1][j] = costDepend[gtasknr-1][j];\n" +
"      }\n" +
"      totalCost[schnr-1][i] += costs[gtasknr-1][i][State]-sharedCostTemp;\n" +
"    }\n" +
"    else {\n" +
"      totalCost[schnr-1][i] += costs[gtasknr-1][i][State];\n" +
"    }\n" +
"  }\n" +
"}\n" +
"\n" +
"//Function that resets the cost in the end of a timeunit\n" +
"void resetCost(int gtasknr, int schnr, int State) {\n" +
"  if (State == READY &amp;&amp; preemptedOrIdle[gtasknr-1]) {\n" +
"    State = PREEMPTED;\n" +
"  }\n" +
"  if (!(State == IDLE &amp;&amp; !preemptedOrIdle[gtasknr-1])) {\n" +
"    for (i :int[0,NOC-1]) {\n" +
"      //Resets the memory_use after first timeunit.\n" +
"      if (i==Memory &amp;&amp; State == RUNNING) {\n" +
"        int sharedCostTemp = 0;\n" +
"        for (j: int[0, MN-1]) {\n" +
"          if (sharedCost[j][gtasknr-1] &gt; 0) {\n" +
"            for (k : int[0, M-1]) {\n" +
"              for (l : int[0,N-1]) {\n" +
"                if (ltog[k][l] == (j+1)) {\n" +
"                  totalCost[k][i] -= sharedCost[j][gtasknr-1];\n" +
"                  sharedCost[j][gtasknr-1] = 0;\n" +
"                }\n" +
"              }\n" +
"            }\n" +
"          }\n" +
"          if (sharedCost[gtasknr-1][j] == costDepend[gtasknr-1][j])\n" +
"            sharedCostTemp -= sharedCost[gtasknr-1][j];\n" +
"          else\n" +
"            sharedCostTemp -= costDepend[gtasknr-1][j];\n" +
"        }\n" +
"        totalCost[schnr-1][i] -= costs[gtasknr-1][i][State]+sharedCostTemp;\n" +
"      }\n" +
"      else {\n" +
"        totalCost[schnr-1][i] -= costs[gtasknr-1][i][State];\n" +
"      }\n" +
"    }\n" +
"  }\n" +
"}\n" +
"\n</declaration><template><name>Task</name><parameter>const int schnr, const int
    gtasknr, const int tasknr, const int emin, const int emax, const int dead,
    const int offset, const int ptime, int prior, int[0,N] &amp;tid, int[0, MN]
    &amp;gtid, chan &amp;ready, chan &amp;run, chan &amp;preempt, chan &amp;finish,
     const int[0,2] sd, int maxCount</parameter><declaration>clock x;\n" +
"int counter = 0;\n" +
"\n" +
"int cp;\n" +
"int cr;\n" +
"\n" +
"int i;\n" +
```

```
"\n" +
"bool taskHasOffset() {\n" +
"  return (offset&gt;0);\n" +
"}\n" +
"\n" +
"bool maxCountReached() {\n" +
"  return (counter&gt;=maxCount &amp;&amp; maxCount &gt; 0);\n" +
"}\n" +
"\n" +
"void updateEndState() {\n" +
"  endState[gtasknr-1]=true;\n" +
"  pending[gtasknr-1] = false;\n" +
"}\n" +
"\n" +
"void firmDeadlineMisses() {\n" +
"  dynamicCriteria[schnr-1][tasknr-1]=0;\n" +
"  Ready[schnr-1][tasknr-1]=false;\n" +
"  Synchronized[gtasknr-1]=false;\n" +
"  Allocated[gtasknr-1]=false;\n" +
"  x = cp;\n" +
"}\n" +
"\n" +
"void finishBeforePeriodEnd() {\n" +
"  tid=tasknr;\n" +
"  gtid = gtasknr;\n" +
"  dynamicCriteria[schnr-1][tasknr-1]=0;\n" +
"  h_fin[schnr-1]=false;\n" +
"  x = cp;\n" +
"  preemptedOrIdle[gtasknr-1] = true;\n"+
"  updateCost(gtasknr, schnr, IDLE);\n" +
"}\n" +
"\n" +
"void finishAtPeriodEnd() {\n" +
"  if (!(sd == FIRM &amp;&amp; cp==ptime &amp;&amp; cr&gt;0)) {\n" +
"    tid=tasknr;\n" +
"    gtid=gtasknr;\n" +
"  }\n" +
"  dynamicCriteria[schnr-1][tasknr-1]=0;\n" +
"  h_fin[schnr-1]=false;\n" +
"  if (counter &lt; maxCount || maxCount == 0) {\n" +
"    pending[gtasknr-1]=true;\n" +
"  }\n" +
"  if (cp &gt; ptime &amp;&amp; sd == SOFT) {\n" +
"    cp = ptime;\n" +
"  }\n" +
"  x=cp;\n" +
"}\n" +
"\n" +
"void runOneTimeUnit() {\n" +
"  h_t[schnr-1]=false;\n" +
"  l_in[schnr-1]=false; \n" +
"\n" +
"  cr--; \n" +
"  cp++;\n" +
"  dynamicCriteria[schnr-1][tasknr-1]--;\n" +
"  nowrun[schnr-1]=false;\n" +
"\n" +
"  if (cr==0 || (sd == FIRM &amp;&amp; cp==dead)) {\n" +
"    h_fin[schnr-1]=true;\n" +
"  }\n" +
"  resetCost(gtasknr, schnr, RUNNING);\n" +
"}\n" +
"\n" +
"bool taskHasMoreRuntime() {\n" +
"  return (cr&gt;0);\n" +
"}\n" +
"\n" +
"bool deadlineMissed() {\n" +
"  return (cp==dead &amp;&amp; sd == HARD &amp;&amp; tid != tasknr);\n" +
"}\n" +
"\n" +
"bool deadlineMissedRun() {\n" +
"  return (cp&gt;dead &amp;&amp; sd == HARD);\n" +
"}\n" +
"\n" +
"bool deadlineNotMissedYetOrSOFT() {\n" +
"  return (((cp&lt;dead &amp;&amp; sd == HARD) \n" +
"          || (sd == SOFT) \n" +
"          || (cp&lt;dead-1 &amp;&amp; sd==FIRM))\n" +
"         &amp;&amp; !locked(controllerLock)\n" +
"         &amp;&amp; !locked(h_fin)\n" +
"         &amp;&amp; !locked(h_t)\n" +
"         &amp;&amp; !pendMN(pending));\n" +
"}\n" +
"\n" +
"bool deadlineNotMissedOrSOFT() {\n" +
"  return (((cp&lt;=dead &amp;&amp; sd == HARD) \n" +
```

```
"            || (sd == SOFT) \n" +
"            || (cp&lt;=dead-1 &amp;&amp; sd==FIRM))\n" +
"             &amp;&amp; tid == tasknr);\n" +
"}\n" +
"\n" +
"bool FirmdeadlineWillMiss() {\n" +
"  return (sd==FIRM\n" +
"           &amp;&amp; cp == (dead-1)\n" +
"           &amp;&amp; tid != tasknr\n" +
"           &amp;&amp; !locked(controllerLock)\n"+
"           &amp;&amp; !locked(h_fin) \n" +
"           &amp;&amp; !locked(h_t));\n" +
"}\n" +
"\n" +
"bool FirmdeadlineMissedBeforePeriod() {\n" +
"  return (sd==FIRM\n" +
"           &amp;&amp; cp == dead\n" +
"           &amp;&amp; ptime &gt; dead\n" +
"           &amp;&amp; cr &gt; 0);\n" +
"}\n" +
"\n" +
"bool FirmdeadlineMissedAtPeriod() {\n" +
"  return (sd==FIRM\n" +
"           &amp;&amp; cp == ptime\n" +
"           &amp;&amp; cr &gt; 0);\n" +
"}\n" +
"\n" +
"void updateMissedDeadline() {\n" +
"  missedDeadline=true;\n" +
"}\n" +
"\n" +
"void setPeriodClock() {\n" +
"  x=ptime-1;\n" +
"}\n" +
"\n" +
"void initOneRun() {\n" +
"  x=0;\n" +
"  l_in[schnr-1]=true;\n" +
"  nowrun[schnr-1]=true;\n" +
"  preemptedOrIdle[gtasknr-1] = false;\n" +
"  updateCost(gtasknr, schnr, RUNNING);\n" +
"}\n" +
"\n" +
"void setPreempted() {\n" +
"  preempted=true;\n" +
"  preemptedOrIdle[gtasknr-1] = true;\n" +
"  x=0;\n" +
"}\n" +
"\n" +
"void initOffset() {\n" +
"  x=0;\n" +
"  updateCost(gtasknr, schnr, STATIC);\n" +
"}\n" +
"\n" +
"void ensureRunCompletion() {\n" +
"  h_t[schnr-1]=true;\n" +
"}\n" +
"\n" +
"bool readyForNextPeriod() {\n" +
"  return !(nowrun[schnr-1] \n" +
"            || locked(h_t)\n" +
"            || locked(h_fin)\n" +
"            || pendMN(h_edf)\n" +
"            || maxCountReached());\n" +
"}\n" +
"\n" +
"void setStaticScheduling() {\n" +
"  if (processorScheduling[schnr-1] == FP) {\n" +
"    staticCriteria[schnr-1][tasknr-1]=prior;  \n" +
"  }\n" +
"  if (processorScheduling[schnr-1] == RM) {\n" +
"    staticCriteria[schnr-1][tasknr-1]=ptime;  \n" +
"  }\n" +
"  if (processorScheduling[schnr-1] == DM) {\n" +
"    staticCriteria[schnr-1][tasknr-1]=dead;  \n" +
"  }\n" +
"}\n" +
"\n" +
"void initNextPeriod(int i) {\n" +
"  cp=0;\n" +
"  tid=tasknr;\n" +
"  pending[gtasknr-1]=false;\n" +
"  dynamicCriteria[schnr-1][tasknr-1]=dead;\n" +
"  setOrigDep(gtasknr-1);\n" +
"  cr=i;\n" +
"  if (!maxCount==0) {\n" +
"    counter++;\n" +
```

```
"   }\n" +
"   setStaticScheduling();\n" +
"\n" +
"\n" +
"   x=0;\n" +
"} \n" +
"\n" +
"void setWaitForUpdateDynamicScheduling() {\n" +
"   h_edf[gtasknr-1]=true;\n" +
"   updateCost(gtasknr, schnr, READY);\n" +
"}\n" +
"\n" +
"void updateDynamicScheduling() {\n" +
"   dynamicCriteria[schnr-1][tasknr-1]--;\n" +
"   x=0;\n" +
"   cp++;\n" +
"   h_edf[gtasknr-1]=false;\n" +
"   resetCost(gtasknr, schnr, READY);\n" +
"}\n" +
"\n" +
"\n" +
"void initNoOffset() {\n" +
"   pending[gtasknr-1]=true;\n" +
"   updateCost(gtasknr, schnr, STATIC);\n" +
"\n" +
"   x=ptime;\n" +
"}\n" +
"\n" +
"void setPend() {\n" +
"   if (counter &lt; maxCount || maxCount == 0) { \n" +
"     pending[gtasknr-1]=true;\n" +
"   }\n" +
"   resetCost(gtasknr, schnr, IDLE);\n" +
"   preemptedOrIdle[gtasknr-1] = false;\n"+
"}\n" +
"\n" +
"bool taskFinishBeforePeriodEnd() {\n" +
"   return (cr==0\n" +
"           &amp;&amp; !locked(h_t)\n" +
"           &amp;&amp; !pendMN(h_edf)\n" +
"           &amp;&amp; cp&lt;ptime\n" +
"           &amp;&amp; (cp&lt;=dead \n" +
"                 || (cp&gt;dead &amp;&amp; sd==SOFT)));\n" +
"}\n" +
"\n" +
"bool taskFinishAtPeriodEnd() {\n" +
"   return (cr==0\n" +
"           &amp;&amp; !locked(h_t)\n" +
"           &amp;&amp; !pendMN(h_edf)\n" +
"           &amp;&amp; ((cp==ptime &amp;&amp;cp==dead) \n" +
"                   || (cp &gt; ptime &amp;&amp; sd == SOFT)));\n" +
"}\n" +
"\n" +
"bool readyForNextRun() {\n" +
"   return (cr&gt;0 \n" +
"           &amp;&amp; !pendMN(pending)\n" +
"           &amp;&amp; !locked(h_t)\n" +
"           &amp;&amp; !locked(h_fin)\n" +
"           &amp;&amp; !locked(controllerLock)\n" +
"           &amp;&amp; !h_r\n" +
"           &amp;&amp; ((sd == SOFT) \n" +
"                 || (cp &lt; dead &amp;&amp; sd == FIRM) \n" +
"                 || (cp &lt;=dead &amp;&amp; sd == HARD))); \n" +
"//         &amp;&amp; !pendMN(h_edf));\n" +
"}\n" +
"\n" +
"</declaration><location id=\"id0\" x=\"-320\" y=\"-424\"><name x=\"-304\"
      y=\"-408\">Finish</name></location><location id=\"id1\" x=\"-520\"
      y=\"-96\"><name x=\"-568\" y=\"-128\">Offset</name><label kind=\"invariant\"
      x=\"-568\" y=\"-80\">//Wait for offset\n" +
"x&lt;=offset-1</label></location><location id=\"id2\" x=\"-520\" y=\"-256\"><name
      x=\"-560\" y=\"-288\">Start</name><committed/></location><location id=\"id3\"
      x=\"128\" y=\"-416\"><name x=\"80\" y=\"-448\">ReadyDynamic</name><label
      kind=\"invariant\" x=\"112\" y=\"-400\">x&lt;=1</label></location><location
      id=\"id4\" x=\"120\" y=\"352\"><name x=\"40\" y=\"336\">Running3</name><label
      kind=\"invariant\" x=\"110\" y=\"367\">x&lt;=1</label></location><location
      id=\"id5\" x=\"200\" y=\"264\"><name x=\"208\" y=\"240\">Running2</name><label
      kind=\"invariant\" x=\"208\" y=\"272\">x&lt;1</label></location><location
      id=\"id6\" x=\"376\" y=\"-256\"><name x=\"384\"
      y=\"-248\">MissedDeadline</name></location><location id=\"id7\" x=\"-320\"
      y=\"-96\"><name x=\"-384\" y=\"-96\">IdleWait</name><label kind=\"invariant\"
      x=\"-400\" y=\"-72\">//Waiting for next period\n" +
"x&lt;ptime</label></location><location id=\"id8\" x=\"128\" y=\"160\"><name
      x=\"48\" y=\"136\">Running</name><urgent/></location><location id=\"id9\"
      x=\"128\" y=\"-256\"><name x=\"118\"
      y=\"-286\">Ready</name><urgent/></location><location id=\"id10\" x=\"-320\"
      y=\"-256\"><name x=\"-352\" y=\"-280\">Idle</name><label kind=\"invariant\"
```

```
      x=\"-428\" y=\"-248\">//Waiting for next period\n" +
"x&lt;=ptime</label></location><init ref=\"id2\"/><transition><source
      ref=\"id8\"/><target ref=\"id6\"/><label kind=\"guard\" x=\"344\"
      y=\"184\">deadlineMissedRun()</label><label kind=\"assignment\" x=\"344\"
      y=\"200\">updateMissedDeadline()</label><nail x=\"376\"
      y=\"160\"/></transition><transition><source ref=\"id9\"/><target
      ref=\"id7\"/><label kind=\"guard\" x=\"-176\"
      y=\"-248\">FirmdeadlineWillMiss()</label><label kind=\"assignment\" x=\"-176\"
      y=\"-232\">firmDeadlineMisses()</label><nail x=\"88\" y=\"-208\"/><nail
      x=\"-250\" y=\"-208\"/></transition><transition><source ref=\"id10\"/><target
      ref=\"id0\"/><label kind=\"guard\" x=\"-456\"
      y=\"-400\">maxCountReached()</label><label kind=\"assignment\" x=\"-432\"
      y=\"-384\">updateEndState()</label></transition><transition><source
      ref=\"id1\"/><target ref=\"id7\"/><label kind=\"guard\" x=\"-496\"
      y=\"-152\">//Finished waiting for offset\n" +
"x==offset-1</label><label kind=\"assignment\" x=\"-496\"
      y=\"-120\">setPeriodClock()</label></transition><transition><source
      ref=\"id2\"/><target ref=\"id1\"/><label kind=\"guard\" x=\"-624\"
      y=\"-240\">taskHasOffset()</label><label kind=\"assignment\" x=\"-656\"
      y=\"-224\">initOffset()" +
"</label></transition><transition><source ref=\"id2\"/><target ref=\"id10\"/><label
      kind=\"guard\" x=\"-488\" y=\"-312\">!taskHasOffset()</label><label
      kind=\"assignment\" x=\"-488\" y=\"-296\">initNoOffset()" +
"</label></transition><transition><source ref=\"id9\"/><target ref=\"id3\"/><label
      kind=\"guard\" x=\"184\" y=\"-384\">deadlineNotMissedYetOrSOFT()</label><label
      kind=\"assignment\" x=\"184\" y=\"-352\">setWaitForUpdateDynamicScheduling()" +
"</label><nail x=\"184\" y=\"-344\"/></transition><transition><source
      ref=\"id3\"/><target ref=\"id9\"/><label kind=\"guard\" x=\"-104\"
      y=\"-376\">//Time unit completion\n" +
"x==1</label><label kind=\"assignment\" x=\"-104\"
      y=\"-344\">updateDynamicScheduling()" +
"</label><nail x=\"72\" y=\"-344\"/></transition><transition><source
      ref=\"id4\"/><target ref=\"id8\"/><label kind=\"guard\" x=\"-72\"
      y=\"224\">//Complete one run\n" +
"x==1</label><label kind=\"assignment\" x=\"-72\" y=\"256\">runOneTimeUnit()" +
"</label><nail x=\"40\" y=\"256\"/></transition><transition><source
      ref=\"id5\"/><target ref=\"id4\"/><label kind=\"guard\" x=\"168\"
      y=\"296\">//Continue run\n" +
"x&gt;0</label><label kind=\"assignment\" x=\"168\"
      y=\"328\">ensureRunCompletion()</label></transition><transition><source
      ref=\"id8\"/><target ref=\"id5\"/><label kind=\"guard\" x=\"176\"
      y=\"168\">readyForNextRun()</label><label kind=\"assignment\" x=\"176\"
      y=\"184\">initOneRun()" +
"</label></transition><transition><source ref=\"id9\"/><target ref=\"id6\"/><label
      kind=\"guard\" x=\"192\" y=\"-296\">deadlineMissed()</label><label
      kind=\"assignment\" x=\"192\"
      y=\"-280\">updateMissedDeadline()</label></transition><transition><source
      ref=\"id8\"/><target ref=\"id10\"/><label kind=\"guard\" x=\"-88\"
      y=\"-50\">taskFinishAtPeriodEnd() ||\n" +
"FirmdeadlineMissedAtPeriod()</label><label kind=\"synchronisation\" x=\"-88\"
      y=\"-24\">finish!</label><label kind=\"assignment\" x=\"-88\"
      y=\"-8\">finishAtPeriodEnd()</label></transition><transition><source
      ref=\"id7\"/><target ref=\"id10\"/><label kind=\"guard\" x=\"-392\"
      y=\"-208\">x&gt;ptime-1</label><label kind=\"assignment\" x=\"-384\"
      y=\"-192\">setPend()</label></transition><transition><source
      ref=\"id8\"/><target ref=\"id7\"/><label kind=\"guard\" x=\"-312\"
      y=\"16\">taskFinishBeforePeriodEnd() ||\n" +
"FirmdeadlineMissedBeforePeriod()</label><label kind=\"synchronisation\" x=\"-312\"
      y=\"48\">finish!</label><label kind=\"assignment\" x=\"-312\"
      y=\"64\">finishBeforePeriodEnd()</label><nail x=\"-32\"
      y=\"160\"/></transition><transition><source ref=\"id8\"/><target
      ref=\"id9\"/><label kind=\"guard\" x=\"200\"
      y=\"-56\">taskHasMoreRuntime()</label><label kind=\"synchronisation\" x=\"200\"
       y=\"-40\">preempt?</label><label kind=\"assignment\" x=\"200\"
      y=\"-24\">setPreempted()</label><nail x=\"192\" y=\"120\"/><nail x=\"192\"
      y=\"-208\"/></transition><transition><source ref=\"id9\"/><target
      ref=\"id8\"/><label kind=\"guard\" x=\"-40\"
      y=\"-176\">deadlineNotMissedOrSOFT()</label><label kind=\"synchronisation\"
      x=\"-40\" y=\"-160\">run?</label></transition><transition><source
      ref=\"id10\"/><target ref=\"id9\"/><label kind=\"select\" x=\"-296\"
      y=\"-344\">i:int[emin,emax]</label><label kind=\"guard\" x=\"-296\"
      y=\"-328\">x==ptime &amp;&amp;\nreadyForNextPeriod()</label><label
      kind=\"synchronisation\" x=\"-296\" y=\"-296\">ready!</label><label
      kind=\"assignment\" x=\"-296\"
      y=\"-280\">initNextPeriod(i)</label></transition></template>" +
"<template><name>Scheduler</name><parameter>const int schnr, int[0,MN] &amp;ftid,
      chan &amp;schedule</parameter><declaration>int[0,N] it; //iteration
      variable\n" +
"int lcri; //variable used to hold the criterion of the task currently chosen\n" +
"int[0,MN] li; //variable used to hold global id for tasks\n" +
"int transID=0;\n" +
"\n" +
"bool aTaskIsReady;\n" +
"\n" +
"void noTasksReady() {\n" +
"  curtid[schnr-1]=0;\n" +
"}\n" +
```

```
"\n" +
"void findHighestPriority() {\n" +
"  for (it : int[0,N-1]) {\n" +
"    li = ltog[schnr-1][it];\n" +
"    if (li!=0) {\n" +
"      if (Allocated[li-1]) {\n" +
"        if (processorScheduling[schnr-1] != EDF &amp;&amp;
    (staticCriteria[schnr-1][it] &lt; lcri)) {\n" +
"          curtid[schnr-1]=it+1;\n" +
"          lcri=staticCriteria[schnr-1][it];\n" +
"        }\n" +
"        else if (processorScheduling[schnr-1] == EDF &amp;&amp;
    (dynamicCriteria[schnr-1][it] &lt; lcri)) {\n" +
"          curtid[schnr-1]=it+1;\n" +
"          lcri=dynamicCriteria[schnr-1][it];\n" +
"        }\n" +
"      }\n" +
"    }\n" +
"  }\n" +
"}\n" +
"\n" +
"void tasksReadyOnProc() {\n" +
"  aTaskIsReady = false;\n" +
"  for (it : int[0,N-1]) {\n" +
"    li = ltog[schnr-1][it];\n" +
"    if (li != 0) {\n" +
"      if (Allocated[li-1]) {\n" +
"        aTaskIsReady = true;\n" +
"          if (processorScheduling[schnr-1] != EDF) {\n" +
"            lcri = staticCriteria[schnr-1][it];\n" +
"          }\n" +
"          else {\n" +
"            lcri = dynamicCriteria[schnr-1][it];\n" +
"          }\n" +
"          curtid[schnr-1] = it+1;\n" +
"          return;\n" +
"      }\n" +
"    }\n" +
"  }\n" +
"}\n" +
"\n" +
"</declaration><location id=\"id11\" x=\"-224\" y=\"-160\"></location><location
    id=\"id12\" x=\"24\" y=\"-160\"><committed/></location><init
    ref=\"id11\"/><transition><source ref=\"id12\"/><target ref=\"id11\"/><label
    kind=\"guard\" x=\"-160\" y=\"-96\">aTaskIsReady</label><label
    kind=\"synchronisation\" x=\"-160\" y=\"-64\">//Scheduler finishing\n" +
"schedule!</label><label kind=\"assignment\" x=\"-160\"
    y=\"-80\">findHighestPriority()</label><nail x=\"-8\" y=\"-96\"/><nail
    x=\"-192\" y=\"-96\"/></transition><transition><source ref=\"id12\"/><target
    ref=\"id11\"/><label kind=\"guard\" x=\"-160\"
    y=\"-296\">!aTaskIsReady</label><label kind=\"synchronisation\" x=\"-160\"
    y=\"-264\">//Scheduler finishing\n" +
"schedule!</label><label kind=\"assignment\" x=\"-160\"
    y=\"-280\">noTasksReady()</label><nail x=\"-8\" y=\"-224\"/><nail x=\"-192\"
    y=\"-224\"/></transition><transition><source ref=\"id11\"/><target
    ref=\"id12\"/><label kind=\"synchronisation\" x=\"-184\" y=\"-216\">//Scheduler
     starting\n" +
"schedule?</label><label kind=\"assignment\" x=\"-184\"
    y=\"-184\">tasksReadyOnProc()</label></transition></template>" +
"<template><name>Control</name><parameter>const int schnr, int[0,MN] &amp;ftid,
    int[0,N] &amp;tid, int[0,MN] &amp;gtid, chan &amp;synchronize, chan
    &amp;schedule, chan &amp;allocate, chan &amp;ready, chan &amp;finish, chan
    &amp;run, chan &amp;preempt</parameter><declaration>int[-1,MN] trigID = -1;\n"
    +
"int lt = 0;\n" +
"\n" +
"void triggers() {\n" +
"  trigID = -1;\n" +
"  for (it:int[0,MN-1]) {\n" +
"    lt = ltog[schnr-1][tid-1];\n" +
"    if (lt != 0) {\n" +
"      if (origdep[it][lt-1] &amp;&amp; taskReadyForTriggering[it]) {\n" +
"        trigID = it;\n" +
"      }\n" +
"    }\n" +
"  }\n" +
"}\n" +
"\n" +
"bool triggerTask() {\n" +
"  return (trigID&gt;(-1));\n" +
"}\n" +
"\n" +
"bool allSignalsReceived() {\n" +
"  return (!pendMN(pending) &amp;&amp; \n" +
"          !locked(h_fin) &amp;&amp; \n" +
"          !pendMN(h_edf));\n" +
"}\n" +
```

```
"\n" +
"void stopProcessor() {\n" +
"  ftid=gtid;\n" +
"  running[schnr-1]=false;\n" +
"}\n" +
"\n" +
"void lockProcessor() {\n" +
"  controllerLock[schnr-1]=true;\n" +
"  processing[schnr-1] = true;\n" +
"}\n" +
"\n" +
"void unlockProcessing() {\n" +
"  processing[schnr-1]=false;\n" +
"}\n" +
"\n" +
"void unlockControllerLock() {\n" +
"  controllerLock[schnr-1]=false;\n" +
"}\n" +
"\n" +
"void setReadyTaskId() {\n" +
"  Ready[schnr - 1][tid - 1]=true;\n" +
"  ftid=0;\n" +
"}\n" +
"\n" +
"bool noReschedule() {\n" +
"  return (!h_r &amp;&amp; !locked(processing));\n" +
"}\n" +
"\n" +
"void setReadyTaskId2() {\n" +
"  Ready[schnr - 1][tid - 1]=true;\n" +
"}\n" +
"\n" +
"bool processorNotRunning() {\n" +
"  return (!running[schnr-1] &amp;&amp; curtid[schnr-1] != 0);\n" +
"}\n" +
"\n" +
"void runProcessor() {\n" +
"  running[schnr-1] = true;\n" +
"}\n" +
"\n" +
"void setRunningTaskId() {\n" +
"  tid=curtid[schnr - 1];\n" +
"  ltid[schnr-1] = curtid[schnr - 1];\n" +
"}\n" +
"\n" +
"bool noSchedulingChange() {\n" +
"  return ((ltid[schnr-1]==curtid[schnr-1] &amp;&amp; running[schnr-1]) ||
    curtid[schnr-1]==0);\n" +
"}\n" +
"\n" +
"bool runningTaskHasLowerPriority() {\n" +
"  return (ltid[schnr-1]!=curtid[schnr-1] &amp;&amp; running[schnr-1]);\n" +
"}\n" +
"\n" +
"void reserve(int task){\n" +
"  for(it:int[0,R-1]){\n" +
"    if (NeededResources[schnr - 1][task - 1][it])\n" +
"      UsedResources[schnr - 1][it]=task;\n" +
"  }\n" +
"}\n" +
"\n" +
"void release(int task){\n" +
"  for(it:int[0,R-1]){\n" +
"    if (NeededResources[schnr - 1][task - 1][it])\n" +
"      UsedResources[schnr - 1][it]=false;\n" +
"  }\n" +
"}\n" +
"</declaration><location id=\"id13\" x=\"-416\" y=\"-304\"
    color=\"#ffffff\"><committed/></location><location id=\"id14\" x=\"-512\"
    y=\"-304\" color=\"#ffffff\"><committed/></location><location id=\"id15\"
    x=\"-32\" y=\"0\" color=\"#ffffff\"><committed/></location><location
    id=\"id16\" x=\"160\" y=\"0\"
    color=\"#ffffff\"><committed/></location><location id=\"id17\" x=\"-128\"
    y=\"-128\"><committed/></location><location id=\"id18\" x=\"-320\"
    y=\"0\"><committed/></location><location id=\"id19\" x=\"-320\"
    y=\"128\"><urgent/></location><location id=\"id20\" x=\"-128\"
    y=\"0\"><committed/></location><location id=\"id21\" x=\"-320\"
    y=\"-128\"><committed/></location><location id=\"id22\" x=\"-128\"
    y=\"-256\"><committed/></location><location id=\"id23\" x=\"-448\" y=\"-256\"
    color=\"#ffffff\"><urgent/></location><location id=\"id24\" x=\"-704\"
    y=\"-256\" color=\"#ffffff\"></location><init ref=\"id24\"/><transition><source
     ref=\"id13\"/><target ref=\"id23\"/><label kind=\"synchronisation\" x=\"-424\"
     y=\"-288\">trigger[trigID]!</label></transition><transition><source
    ref=\"id14\"/><target ref=\"id13\"/><label kind=\"guard\" x=\"-496\"
    y=\"-328\">triggerTask()</label></transition><transition><source
    ref=\"id14\"/><target ref=\"id23\"/><label kind=\"guard\" x=\"-552\"
    y=\"-264\">!triggerTask()</label><nail x=\"-496\"
```

```
        y=\"-256\"/></transition><transition><source ref=\"id19\"/><target
        ref=\"id15\"/><label kind=\"guard\" x=\"-280\"
        y=\"104\">noReschedule()</label><nail x=\"-256\" y=\"128\"/><nail x=\"-32\"
        y=\"128\"/></transition><transition><source ref=\"id19\"/><target
        ref=\"id21\"/><label kind=\"synchronisation\" x=\"-432\"
        y=\"32\">reschedule?</label><nail x=\"-352\" y=\"128\"/><nail x=\"-352\"
        y=\"-64\"/></transition><transition><source ref=\"id16\"/><target
        ref=\"id24\"/><label kind=\"synchronisation\" x=\"128\"
        y=\"-328\">run!</label><label kind=\"assignment\" x=\"64\"
        y=\"-312\">runProcessor(),\n" +
"reserve(tid),\n" +
"unlockControllerLock()</label><nail x=\"160\" y=\"-384\"/><nail x=\"-736\"
        y=\"-384\"/><nail x=\"-736\" y=\"-256\"/></transition><transition><source
        ref=\"id15\"/><target ref=\"id24\"/><label kind=\"guard\" x=\"-32\"
        y=\"-208\">noSchedulingChange()</label><label kind=\"assignment\" x=\"-32\"
        y=\"-192\">unlockControllerLock()</label><nail x=\"-32\" y=\"-352\"/><nail
        x=\"-704\" y=\"-352\"/></transition><transition><source ref=\"id15\"/><target
        ref=\"id16\"/><label kind=\"guard\" x=\"8\"
        y=\"-72\">processorNotRunning()</label><label kind=\"assignment\" x=\"8\"
        y=\"-56\">setRunningTaskId()</label><nail x=\"0\" y=\"-32\"/><nail x=\"128\"
        y=\"-32\"/></transition><transition><source ref=\"id17\"/><target
        ref=\"id18\"/><label kind=\"synchronisation\" x=\"-264\"
        y=\"-80\">allocate?</label><nail x=\"-128\" y=\"-64\"/><nail x=\"-320\"
        y=\"-64\"/></transition><transition><source ref=\"id21\"/><target
        ref=\"id17\"/><label kind=\"synchronisation\" x=\"-264\"
        y=\"-144\">allocate!</label></transition><transition><source
        ref=\"id15\"/><target ref=\"id16\"/><label kind=\"guard\" x=\"-16\"
        y=\"32\">runningTaskHasLowerPriority()</label><label kind=\"synchronisation\"
        x=\"-16\" y=\"48\">preempt!</label><label kind=\"assignment\" x=\"-16\"
        y=\"64\">setRunningTaskId()</label><nail x=\"0\" y=\"32\"/><nail x=\"128\"
        y=\"32\"/></transition><transition><source ref=\"id23\"/><target
        ref=\"id23\"/><label kind=\"synchronisation\" x=\"-504\" y=\"-208\">//Wait for
        all Ready signals\n" +
"ready?</label><label kind=\"assignment\" x=\"-504\"
        y=\"-176\">setReadyTaskId2()</label><nail x=\"-400\" y=\"-208\"/><nail
        x=\"-448\" y=\"-208\"/></transition><transition><source ref=\"id24\"/><target
        ref=\"id21\"/><label kind=\"synchronisation\" x=\"-632\"
        y=\"-128\">reschedule?</label><label kind=\"assignment\" x=\"-632\"
        y=\"-112\">lockProcessor()</label><nail x=\"-704\" y=\"-128\"/><nail x=\"-352\"
         y=\"-128\"/></transition><transition><source ref=\"id20\"/><target
        ref=\"id19\"/><label kind=\"synchronisation\" x=\"-272\" y=\"32\">//Wait for
        scheduler\n" +
"schedule?</label><label kind=\"assignment\" x=\"-272\"
        y=\"64\">unlockProcessing()</label><nail x=\"-128\" y=\"64\"/><nail x=\"-320\"
        y=\"64\"/></transition><transition><source ref=\"id18\"/><target
        ref=\"id20\"/><label kind=\"synchronisation\" x=\"-264\" y=\"-32\">//Activate
        scheduler\n" +
"schedule!</label></transition><transition><source ref=\"id22\"/><target
        ref=\"id21\"/><label kind=\"synchronisation\" x=\"-280\" y=\"-224\">//Wait for
        synchronizer\n" +
"synchronize?</label><nail x=\"-128\" y=\"-192\"/><nail x=\"-320\"
        y=\"-192\"/></transition><transition><source ref=\"id23\"/><target
        ref=\"id22\"/><label kind=\"guard\" x=\"-288\"
        y=\"-304\">allSignalsReceived()</label><label kind=\"synchronisation\"
        x=\"-288\" y=\"-288\">//activate synchronizer\n" +
"synchronize!</label></transition><transition><source ref=\"id24\"/><target
        ref=\"id23\"/><label kind=\"guard\" x=\"-696\"
        y=\"-240\">!h_fin[schnr-1]</label><label kind=\"synchronisation\" x=\"-696\"
        y=\"-224\">//Get first ready signal\n" +
"ready?</label><label kind=\"assignment\" x=\"-696\"
        y=\"-192\">setReadyTaskId(),\n" +
"lockProcessor()</label><nail x=\"-672\" y=\"-224\"/><nail x=\"-480\"
        y=\"-224\"/></transition><transition><source ref=\"id24\"/><target
        ref=\"id14\"/><label kind=\"synchronisation\" x=\"-640\" y=\"-336\">//Get one
        finish signal\n" +
"finish?</label><label kind=\"assignment\" x=\"-640\"
        y=\"-304\">stopProcessor(),\n" +
"lockProcessor(),\n" +
"release(tid),\n" +
"triggers()</label><nail x=\"-672\" y=\"-304\"/></transition></template>" +
"<template><name>Synchronizer</name><parameter>const int schnr, int[0,MN] &amp;ftid,
         chan &amp;synchronize</parameter><declaration>int[0,MN] i; //iteration
        variable\n" +
"//int[0,M*N] it; //iteration variable\n" +
"int[0,MN] li; //variable used to hold global id for tasks\n" +
"bool depCh; //flag used if a dependency has been changed\n" +
"\n" +
"\n" +
"bool aDependencyHasChanged() {\n" +
"  return (depCh);\n" +
"}\n" +
"\n" +
"bool aTaskHasFinished() {\n" +
"  return (ftid!=0);\n" +
"}\n" +
"\n" +
"void setGlobalReschedule() {\n" +
```

```
"   depCh=false;\n" +
"   h_r = true;\n" +
"}\n" +
"\n" +
"\n" +
"\n" +
"void syncFinish() {\n" +
"   if (ftid &gt; 0) {\n" +
"     opdDep(ftid-1);\n" +
"     Synchronized[ftid-1]=false;\n" +
"     for (i : int[0,MN-1]) {\n" +
"       if (WaitDep[i] &amp;&amp; !taskHasDependency(i)) {\n" +
"         Synchronized[i]=true;\n" +
"         WaitDep[i]=false; \n" +
"         depCh=true;\n" +
"       }\n" +
"     }\n" +
"   }\n" +
"}\n" +
"\n" +
"void syncReady() {\n" +
"   for (i : int[0, N-1]) {\n" +
"     if (Ready[schnr-1][i]) {\n" +
"       li=ltog[schnr-1][i];\n" +
"       Ready[schnr-1][i]=false;\n" +
"       if (taskHasDependency(li-1)) {\n" +
"         WaitDep[li-1]=true;\n" +
"       }\n" +
"       else {\n" +
"         Synchronized[li-1]=true;\n" +
"       }\n" +
"     }\n" +
"   }\n" +
"}</declaration><location id=\"id25\" x=\"32\"
      y=\"-192\"><committed/></location><location id=\"id26\" x=\"-224\"
      y=\"-192\"><committed/></location><location id=\"id27\" x=\"-224\"
      y=\"-40\"></location><init ref=\"id27\"/><transition><source
      ref=\"id26\"/><target ref=\"id27\"/><label kind=\"synchronisation\" x=\"-384\"
      y=\"-120\">//Synchronizer finishing\n" +
"synchronize!</label></transition><transition><source ref=\"id25\"/><target
      ref=\"id27\"/><label kind=\"guard\" x=\"-200\"
      y=\"-152\">!aDependencyHasChanged()</label><label kind=\"synchronisation\"
      x=\"-200\" y=\"-136\">//Synchronizer finishing\n" +
"synchronize!</label></transition><transition><source ref=\"id25\"/><target
      ref=\"id26\"/><label kind=\"guard\" x=\"-160\"
      y=\"-248\">aDependencyHasChanged()</label><label kind=\"synchronisation\"
      x=\"-160\" y=\"-232\">activateReschedule!</label><label kind=\"assignment\"
      x=\"-160\"
      y=\"-216\">setGlobalReschedule()</label></transition><transition><source
      ref=\"id27\"/><target ref=\"id25\"/><label kind=\"synchronisation\" x=\"40\"
      y=\"-136\">//Synchronizer starting\n" +
"synchronize?</label><label kind=\"assignment\" x=\"40\"
      y=\"-104\">syncFinish(),\n" +
"syncReady()</label><nail x=\"32\" y=\"-40\"/></transition></template>" +
"<template><name>Allocator</name><parameter>const int schnr, int allocationProtocol,
       chan &amp;allocate</parameter><declaration>int gti;\n" +
"bool resourceConflict,sendResched;\n" +
"\n" +
"void removeReschedFlag() {\n" +
"   sendResched=false;\n" +
"}\n" +
"\n" +
"void checkResources(){\n" +
"   for(it:int[0,N-1]){\n" +
"     gti=ltog[schnr-1][it];\n" +
"     if (gti){\n" +
"       if(Synchronized[gti-1]){\n" +
"         for(r:int[0,R-1]){\n" +
"           if (UsedResources[schnr-1][r] &amp;&amp; allocationProtocol==NPCS) {\n" +
"             resourceConflict = true;\n" +
"           }\n" +
"           else if(NeededResources[schnr-1][it][r] &amp;&amp;
      UsedResources[schnr-1][r]){\n" +
"             resourceConflict=true;\n" +
"             if(allocationProtocol==PRI_INH &amp;&amp;
      (staticCriteria[schnr-1][(UsedResources[schnr-1][r])-1] &gt;
      staticCriteria[schnr-1][it])){\n" +
"               staticCriteria[schnr-1][(UsedResources[schnr-1][r])-1] =
      staticCriteria[schnr-1][it];\n" +
"               h_r=true;\n" +
"               sendResched=true;\n" +
"             }\n" +
"             if(allocationProtocol==PRI_INH &amp;&amp;
      (dynamicCriteria[schnr-1][(UsedResources[schnr-1][r])-1] &gt;
      dynamicCriteria[schnr-1][it])){\n" +
"               dynamicCriteria[schnr-1][(UsedResources[schnr-1][r])-1] =
      dynamicCriteria[schnr-1][it];\n" +
```

```
"              h_r=true;\n" +
"              sendResched=true;\n" +
"            }\n" +
"          }\n" +
"        }\n" +
"        if(!resourceConflict)\n" +
"          Allocated[gti-1]=true;\n" +
"        resourceConflict=false;\n" +
"      }\n" +
"      else{\n" +
"        Allocated[gti-1]=false;\n" +
"      }\n" +
"    }\n" +
"  }\n" +
"}\n" +
"</declaration><location id=\"id28\" x=\"-192\"
    y=\"-160\"><committed/></location><location id=\"id29\" x=\"-352\"
    y=\"-160\"></location><init ref=\"id29\"/><transition><source
    ref=\"id28\"/><target ref=\"id28\"/><label kind=\"guard\" x=\"-120\"
    y=\"-184\">sendResched</label><label kind=\"synchronisation\" x=\"-120\"
    y=\"-168\">activateReschedule!</label><label kind=\"assignment\" x=\"-120\"
    y=\"-152\">removeReschedFlag()</label><nail x=\"-128\" y=\"-192\"/><nail
    x=\"-128\" y=\"-128\"/></transition><transition><source ref=\"id28\"/><target
    ref=\"id29\"/><label kind=\"guard\" x=\"-320\"
    y=\"-128\">!sendResched</label><label kind=\"synchronisation\" x=\"-320\"
    y=\"-112\">allocate!</label><nail x=\"-224\" y=\"-128\"/><nail x=\"-320\"
    y=\"-128\"/></transition><transition><source ref=\"id29\"/><target
    ref=\"id28\"/><label kind=\"synchronisation\" x=\"-312\"
    y=\"-232\">allocate?</label><label kind=\"assignment\" x=\"-312\"
    y=\"-216\">checkResources()</label><nail x=\"-320\" y=\"-192\"/><nail
    x=\"-224\" y=\"-192\"/></transition></template>" +
"<template><name>Rescheduler</name><declaration>void unlockReschedule() {\n" +
"  h_r = false;\n" +
"}</declaration><location id=\"id30\" x=\"32\"
    y=\"-96\"><urgent/></location><location id=\"id31\" x=\"-192\"
    y=\"-96\"></location><init ref=\"id31\"/><transition><source
    ref=\"id30\"/><target ref=\"id30\"/><label kind=\"synchronisation\" x=\"104\"
    y=\"-112\">activateReschedule?</label><nail x=\"96\" y=\"-128\"/><nail x=\"96\"
    y=\"-64\"/></transition><transition><source ref=\"id30\"/><target
    ref=\"id31\"/><label kind=\"guard\" x=\"-136\"
    y=\"-64\">notSignalsPending()</label><label kind=\"synchronisation\" x=\"-136\"
    y=\"-48\">reschedule!</label><label kind=\"assignment\" x=\"-136\"
    y=\"-32\">unlockReschedule()</label><nail x=\"0\" y=\"-64\"/><nail x=\"-160\"
    y=\"-64\"/></transition><transition><source ref=\"id31\"/><target
    ref=\"id30\"/><label kind=\"synchronisation\" x=\"-144\"
    y=\"-152\">activateReschedule?</label><nail x=\"-160\" y=\"-128\"/><nail
    x=\"0\" y=\"-128\"/></transition></template>" +
"<template><name>Triggered</name><parameter>const int schnr, const int gtasknr,
    const int tasknr, const int emin, const int emax, const int dead, int prior,
    int[0,N] &amp;tid, int[0,MN] &amp;gtid,chan &amp;trigger, chan &amp;ready, chan
    &amp;run, chan &amp;preempt, chan &amp;finish, const int[0,2]
    sd</parameter><declaration>clock x;\n" +
"int counter = 0;\n" +
"\n" +
"int cp;\n" +
"int cr;\n" +
"\n" +
"int i;\n" +
"\n" +
"void setReadyForTriggering() {\n" +
"  taskReadyForTriggering[gtasknr-1] = true;\n" +
"  endState[gtasknr-1] = true;\n" +
"  updateCost(gtasknr, schnr, STATIC);\n" +
"  resetCost(gtasknr, schnr, IDLE);\n" +
"  preemptedOrIdle[gtasknr-1] = false;\n" +
"}\n" +
"\n" +
"void finished() {\n" +
"  if (!(sd==FIRM &amp;&amp; cp==dead &amp;&amp; cr &gt; 0)) {\n" +
"    tid=tasknr;\n" +
"    gtid=gtasknr;\n" +
"  }\n" +
"  endState[gtasknr-1]=true;\n" +
"  h_fin[schnr-1] = false;\n" +
"  updateCost(gtasknr, schnr, IDLE);\n" +
"  preemptedOrIdle[gtasknr-1] = true;\n" +
"}\n" +
"\n" +
"void assignEnoughRuntime() {\n" +
"  cp=dead-cr;\n" +
"  x=0;\n" +
"}\n" +
"\n" +
"void runOneTimeUnit() {\n" +
"  h_t[schnr-1]=false;\n" +
"  l_in[schnr-1]=false; \n" +
"\n" +
```

```
"   cr--; \n" +
"   cp++;\n" +
"   dynamicCriteria[schnr-1][tasknr-1]--;\n" +
"   nowrun[schnr-1]=false;\n" +
"   resetCost(gtasknr, schnr, RUNNING);\n" +
"   if (cr==0 || (sd == FIRM &amp;&amp; cp==dead)) {\n" +
"      h_fin[schnr-1]=true;\n" +
"   }\n" +
"}\n" +
"\n" +
"bool taskHasMoreRuntime() {\n" +
"   return (cr&gt;0);\n" +
"}\n" +
"\n" +
"bool deadlineMissed() {\n" +
"   return (cp==dead &amp;&amp; sd == HARD &amp;&amp; tid != tasknr);\n" +
"}\n" +
"\n" +
"bool deadlineMissedRun() {\n" +
"   return (cp==dead &amp;&amp; sd == HARD &amp;&amp; cr &gt; 0);\n" +
"}\n" +
"\n" +
"bool deadlineNotMissedYetOrSOFT() {\n" +
"   return (((cp&lt;dead &amp;&amp; sd == HARD) \n" +
"             || (sd == SOFT) \n" +
"             || (cp&lt;dead-1 &amp;&amp; sd==FIRM))\n" +
"            &amp;&amp; !locked(controllerLock)\n" +
"            &amp;&amp; !locked(h_fin)\n" +
"            &amp;&amp; !locked(h_t));\n" +
"}\n" +
"\n" +
"bool deadlineNotMissedOrSOFT() {\n" +
"   return (((cp&lt;=dead &amp;&amp; sd == HARD) \n" +
"             || (sd == SOFT) \n" +
"             || (cp&lt;=dead-1 &amp;&amp; sd==FIRM))\n" +
"              &amp;&amp; tid == tasknr);\n" +
"}\n" +
"\n" +
"bool FirmdeadlineWillMiss() {\n" +
"   return (sd==FIRM\n" +
"            &amp;&amp; cp == (dead-1)\n" +
"            &amp;&amp; tid != tasknr);\n" +
"}\n" +
"\n" +
"bool FirmdeadlineMissed() {\n" +
"   return (sd==FIRM &amp;&amp;\n" +
"             cp==dead &amp;&amp;\n" +
"             cr &gt; 0);\n" +
"}\n" +
"\n" +
"void setMissedDeadline() {\n" +
"   missedDeadline=true;\n" +
"}\n" +
"\n" +
"void initOneRun() {\n" +
"   x=0;\n" +
"   l_in[schnr-1]=true;\n" +
"   nowrun[schnr-1]=true;\n" +
"   preemptedOrIdle[gtasknr-1] = false;\n" +
"   updateCost(gtasknr, schnr, RUNNING);\n" +
"}\n" +
"\n" +
"void setPreempted() {\n" +
"   preempted = true;\n" +
"   preemptedOrIdle[gtasknr-1] = true;\n"+
"   x=0;\n" +
"}\n" +
"\n" +
"void ensureRunCompletion() {\n" +
"   h_t[schnr-1]=true;\n" +
"}\n" +
"\n" +
"bool finishSignalsPending() {\n" +
"   return (nowrun[schnr-1] \n" +
"             || locked(h_fin)\n" +
"             || pendMN(h_edf));\n" +
"}\n" +
"\n" +
"void initNextPeriod(int i) {\n" +
"   cp=0;\n" +
"   tid=tasknr;\n" +
"   gtid = gtasknr;\n" +
"   pending[gtasknr-1]=false;\n" +
"   endState[gtasknr-1] = false;\n" +
"   dynamicCriteria[schnr-1][tasknr-1]=dead;\n" +
"   setOrigDep(gtasknr-1);\n" +
"   cr=i;\n" +
```

```
"  x=0;\n" +
"} \n" +
"\n" +
"void setWaitForUpdateDynamicScheduling() {\n" +
"  h_edf[gtasknr-1]=true;\n" +
"  updateCost(gtasknr, schnr, READY);\n" +
"}\n" +
"\n" +
"void updateDynamicScheduling() {\n" +
"  dynamicCriteria[schnr-1][tasknr-1]--;\n" +
"  x=0;\n" +
"  cp++;\n" +
"  h_edf[gtasknr-1]=false;\n" +
"  resetCost(gtasknr, schnr, READY);\n" +
"}\n" +
"\n" +
"\n" +
"void initTask() {\n" +
"  pending[gtasknr-1]=true;\n" +
"  if (processorScheduling[schnr-1] == FP) {\n" +
"    staticCriteria[schnr-1][tasknr-1]=prior;  \n" +
"  }\n" +
"  else {\n" +
"    staticCriteria[schnr-1][tasknr-1]=dead;  \n" +
"  }\n" +
"  taskReadyForTriggering[gtasknr-1] = false;\n" +
"}\n" +
"\n" +
"bool taskHasFinished() {\n" +
"  return (cr==0\n" +
"         &amp;&amp; !locked(h_t)\n" +
"         &amp;&amp; !pendMN(h_edf));\n" +
"}\n" +
"\n" +
"bool readyForNextRun() {\n" +
"  return (cr&gt;0 \n" +
"         &amp;&amp; !pendMN(pending)\n" +
"         &amp;&amp; !locked(h_t)\n" +
"         &amp;&amp; !locked(h_fin)\n" +
"         &amp;&amp; !locked(controllerLock)\n" +
"         &amp;&amp; !h_r\n" +
"         &amp;&amp; ((sd == SOFT) \n" +
"             || (cp &lt; dead &amp;&amp; sd == FIRM) \n" +
"             || (cp &lt;=dead &amp;&amp; sd == HARD))); \n" +
"//        &amp;&amp; !pendMN(h_edf));\n" +
"}\n" +
"\n" +
"void firmDeadlineMisses() {\n" +
"  dynamicCriteria[schnr-1][tasknr-1]=0;\n" +
"  Ready[schnr-1][tasknr-1]=false;\n" +
"  Synchronized[gtasknr-1]=false;\n" +
"  Allocated[gtasknr-1]=false;\n" +
"  updateCost(gtasknr, schnr, IDLE);\n" +
"  preemptedOrIdle[gtasknr-1] = true;\n" +
"  x = cp;\n" +
"}\n" +
"\n" +
"</declaration><location id=\"id32\" x=\"-424\" y=\"-256\"><name x=\"-480\"
    y=\"-288\">IdleWait</name></location><location id=\"id33\" x=\"-296\"
    y=\"-256\"><urgent/></location><location id=\"id34\" x=\"-424\"
    y=\"-128\"><name x=\"-456\"
    y=\"-120\">Start</name><committed/></location><location id=\"id35\" x=\"-24\"
    y=\"-416\"><name x=\"-72\" y=\"-448\">ReadyDynamic</name><label
    kind=\"invariant\" x=\"-40\" y=\"-400\">x&lt;=1</label></location><location
    id=\"id36\" x=\"-32\" y=\"352\"><name x=\"-112\"
    y=\"336\">Running3</name><label kind=\"invariant\" x=\"-42\"
    y=\"367\">x&lt;=1</label></location><location id=\"id37\" x=\"48\"
    y=\"264\"><name x=\"56\" y=\"240\">Running2</name><label kind=\"invariant\"
    x=\"56\" y=\"272\">x&lt;1</label></location><location id=\"id38\" x=\"224\"
    y=\"-256\"><name x=\"232\" y=\"-248\">MissedDeadline</name></location><location
     id=\"id39\" x=\"-24\" y=\"160\"><name x=\"-104\"
    y=\"136\">Running</name><urgent/></location><location id=\"id40\" x=\"-24\"
    y=\"-256\"><name x=\"-34\" y=\"-286\">Ready</name><urgent/></location><init
    ref=\"id34\"/><transition><source ref=\"id34\"/><target ref=\"id32\"/><label
    kind=\"assignment\" x=\"-416\"
    y=\"-192\">setReadyForTriggering()</label></transition><transition><source
    ref=\"id32\"/><target ref=\"id33\"/><label kind=\"synchronisation\" x=\"-392\"
    y=\"-296\">trigger?</label><label kind=\"assignment\" x=\"-392\"
    y=\"-280\">initTask()</label></transition><transition><source
    ref=\"id40\"/><target ref=\"id34\"/><label kind=\"guard\" x=\"-288\"
    y=\"-152\">FirmdeadlineWillMiss()</label><label kind=\"assignment\" x=\"-288\"
    y=\"-128\">firmDeadlineMisses()</label><nail x=\"-80\"
    y=\"-128\"/></transition><transition><source ref=\"id40\"/><target
    ref=\"id35\"/><label kind=\"guard\" x=\"24\" y=\"-380\">" +
"deadlineNotMissedYetOrSOFT()</label><label kind=\"assignment\" x=\"32\"
    y=\"-368\">setWaitForUpdateDynamicScheduling()" +
```

```
"</label><nail x=\"32\" y=\"-344\"/></transition><transition><source
      ref=\"id35\"/><target ref=\"id40\"/><label kind=\"guard\" x=\"-224\"
      y=\"-400\">//Time unit completion\n" +
"x==1</label><label kind=\"assignment\" x=\"-256\"
      y=\"-368\">updateDynamicScheduling()" +
"</label><nail x=\"-80\" y=\"-344\"/></transition><transition><source
      ref=\"id36\"/><target ref=\"id39\"/><label kind=\"guard\" x=\"-224\"
      y=\"224\">//Complete one run\n" +
"x==1</label><label kind=\"assignment\" x=\"-224\" y=\"256\">runOneTimeUnit()" +
"</label><nail x=\"-112\" y=\"256\"/></transition><transition><source
      ref=\"id37\"/><target ref=\"id36\"/><label kind=\"guard\" x=\"16\"
      y=\"296\">//Continue run\n" +
"x&gt;0</label><label kind=\"assignment\" x=\"16\"
      y=\"328\">ensureRunCompletion()</label></transition><transition><source
      ref=\"id39\"/><target ref=\"id37\"/><label kind=\"guard\" x=\"24\"
      y=\"168\">readyForNextRun()</label><label kind=\"assignment\" x=\"24\"
      y=\"184\">initOneRun()" +
"</label></transition><transition><source ref=\"id40\"/><target ref=\"id38\"/><label
       kind=\"guard\" x=\"48\" y=\"-296\">deadlineMissed()</label><label
      kind=\"assignment\" x=\"72\"
      y=\"-276\">setMissedDeadline()</label></transition><transition><source
      ref=\"id39\"/><target ref=\"id38\"/><label kind=\"guard\" x=\"216\"
      y=\"164\">deadlineMissedRun()</label><label kind=\"assignment\" x=\"216\"
      y=\"184\">setMissedDeadline()</label><nail x=\"224\"
      y=\"160\"/></transition><transition><source ref=\"id39\"/><target
      ref=\"id34\"/><label kind=\"guard\" x=\"-304\" y=\"22\">taskHasFinished()
      ||\nFirmdeadlineMissed()</label><label kind=\"synchronisation\" x=\"-304\"
      y=\"48\">finish!</label><label kind=\"assignment\" x=\"-304\"
      y=\"64\">finished()</label></transition><transition><source
      ref=\"id39\"/><target ref=\"id40\"/><label kind=\"guard\" x=\"48\"
      y=\"-80\">taskHasMoreRuntime()</label><label kind=\"synchronisation\" x=\"48\"
      y=\"-64\">preempt?</label><label kind=\"assignment\" x=\"48\"
      y=\"-48\">setPreempted()</label><nail x=\"40\" y=\"128\"/><nail x=\"40\"
      y=\"-216\"/></transition><transition><source ref=\"id40\"/><target
      ref=\"id39\"/><label kind=\"guard\" x=\"-208\" y=\"-188\">" +
"deadlineNotMissedOrSOFT()</label><label kind=\"synchronisation\" x=\"-208\"
      y=\"-176\">run?</label></transition><transition><source ref=\"id33\"/><target
      ref=\"id40\"/><label kind=\"select\" x=\"-256\"
      y=\"-327\">i:int[emin,emax]</label><label kind=\"guard\" x=\"-256\"
      y=\"-312\">!finishSignalsPending()</label><label kind=\"synchronisation\"
      x=\"-256\" y=\"-296\">ready!</label><label kind=\"assignment\" x=\"-256\"
      y=\"-280\">initNextPeriod(i)</label></transition></template>" +
"<template><name>Environment</name><parameter>int task_id, chan &amp; trigger, int
      interarrival</parameter><declaration>int temp;\n" +
"clock x; \n" +
"int cl;\n" +
"\n" +
"bool taskIsReady() {\n"+
"  return (!locked(h_t) \n" +
"          &amp;&amp; !locked(l_in) \n"+
"          &amp;&amp; taskReadyForTriggering[task_id-1] \n"+
"         &amp;&amp; cl &gt;=interarrival);\n"+
"}\n"+
"\n"+
"void updateClock() {\n"+
"  x=0;\n"+
"  if (cl &lt; 32000) {\n" +
"     cl++;\n"+
"  }\n"+
"}\n"+
"\n"+
"void resetClock() {\n"+
"  x=0;\n"+
"  cl=0;\n"+
"}</declaration><location id=\"id41\" x=\"-768\"
      y=\"0\"><committed/></location><location id=\"id42\" x=\"-960\" y=\"0\"><label
      kind=\"invariant\" x=\"-970\" y=\"15\">x&lt;=1</label></location><init
      ref=\"id42\"/><transition><source ref=\"id42\"/><target ref=\"id42\"/><label
      kind=\"guard\" x=\"-1064\" y=\"-16\">x==1</label><label kind=\"assignment\"
      x=\"-1104\" y=\"0\">updateClock()</label><nail x=\"-1024\" y=\"-24\"/><nail
      x=\"-1024\" y=\"24\"/></transition><transition><source ref=\"id41\"/><target
      ref=\"id42\"/><label kind=\"synchronisation\" x=\"-896\"
      y=\"64\">trigger!</label><label kind=\"assignment\" x=\"-896\"
      y=\"80\">resetClock()</label><nail x=\"-864\"
      y=\"32\"/></transition><transition><source ref=\"id42\"/><target
      ref=\"id41\"/><label kind=\"guard\" x=\"-912\" y=\"-80\">taskIsReady()
      &amp;&amp; x==0</label><nail x=\"-864\"
      y=\"-32\"/></transition></template><system>\n");
toStringAll(application.tasks,platform.processors);
toStringSystem(application.tasks,platform.processors);
fout.print(";</system></nta>");
}}
```

E.8 Parser.java

```
/**
 * Parser.java
 * Parser class for MoVES.
 * The Parser class handles the parsing of traces generated by Verifyta.
 *
 * The trace is stored in the String[] taskRun.
 *
 * @author    Kristian Staaloe Knudsen, Jens Ellebaek Nielsen
 * @version   1.1
 * @access    public
 */
package MoVES;

import java.io.*;

class Parser {
    /** Single array containing all tasks */
    private Task[] taskSingle;
    /** Integer */
    private int tn;
    /** Integer array with the clocks used in the parser */
    private int[] clock;
    /** String array containing the schedule for each task */
    private String[] taskRun;

    /** A line from the Verifier class */
    private String strLine;

    /** Which task has just been registered as running? */
    private int runningTaskTemp = 0;
    /** Which task has just been registered as running? */
    private int  runningTask = 0;

    /** has the idle tasks been counted? */
    private Boolean boolCountIdleTasks = false;
    /** reset the clock */
    private Boolean[] nulstil;
    /** Used to read an Input Stream */
    private BufferedInputStream bis;
    /** Used to read an input stream */
    private BufferedReader bur;

    /** Used to read a file */
    private FileReader fir;

    /** How long tine is the to the next run? */
    int timeBeforeNextRun = 0;

   /**
    * Constructor
    * @param t Task[] which contains all tasks in the system
    */
    public Parser(Task[] t) {
        taskSingle = t;
        tn = taskSingle.length;
        clock = new int[tn];

        taskRun = new String[tn];
        nulstil = new Boolean[tn];

        for (int i = 0; i<tn; i++) {
            clock[i] = 0;
            taskRun[i] = "";
            nulstil[i] = false;
            if (taskSingle[i].o <= 0) {
                taskSingle[i].offsetExceeded = true;
            }
        }
    }

   /**
    * Function to test if all tasks ends in the finish-state
    * @param strLine String containing the trace-information.
    * @return boolean True if all tasks ends in the finish state.
    */
    public boolean allFinish(String strLine) {
        for (int i = 0; i<tn; i++)
            if (!taskSingle[i].nonperiodic)
                if (strLine.indexOf("Taskc" + (i+1) + ".Finish")<0)
                    return false;
        return true;
    }
```

```java
/**
 * Parses a trace line.
 * If the line is a unique state where the next state will be in the next
 * time period, the position of tasks is stored, otherwise the line is trown
 *     away.
 * @param strLine String containing a trace-line
 */
public void parseTrace(String strLine) {
    for (int j = 0; j<tn; j++) {
        if (strLine.indexOf("Taskc" + (j+1) + ".Idle")>0 &&
             !taskSingle[j].offsetExceeded) {
            nulstil[j] = true;
        }
        else if (strLine.indexOf("Taskc" + (j+1) + ".Idle ")>0) {
            boolCountIdleTasks = false;
            //check if any task is in idle, if so, the clock is reset
            if (!taskSingle[j].offsetExceeded
                && taskSingle[j].o>0) {
                //If the task has an offset, and it has not exceeded yet.
                nulstil[j] = true;
            }
            clock[j] = 0;
        }
    }
    if (strLine.indexOf(".Running3")>0) {
        //A task is in runningstate.
        if (strLine.indexOf(".Running2") > 0
            || strLine.indexOf(".Ready ") > 0) {
        }
        else {
            //All tasks is in between of two timeunits.
            runningTaskTemp=0;
            for (int j=0; j<tn; j++) {
                if (runningTask == 0) {
                    clock[j]++;
                    if (strLine.indexOf("Taskc" + (j+1) + ".Running3")>0) {
                        if (clock[j]<=taskSingle[j].d ||
                             taskSingle[j].nonperiodic) {
                            //the task is running in its ordinary period
                            taskRun[j] += "1";
                        }
                        else {
                            taskRun[j] += "X";


                        }
                        taskSingle[j].offsetExceeded = true;
                        nulstil[j]=false;
                        runningTaskTemp = (j+1);
                    }
                    else {
                        if (!taskSingle[j].offsetExceeded) {
                            if (nulstil[j]) {
                                taskSingle[j].offsetExceeded = true;
                                nulstil[j]=false;
                            }
                            taskRun[j]+= "-";
                        }
                        else if (strLine.indexOf("Taskc" + (j+1) +
                             ".Finish")>0) {
                          taskRun[j] += "*";
                        }
                        else if (clock[j]<=taskSingle[j].d
                                 || taskSingle[j].nonperiodic
                                 || strLine.indexOf("Taskc" + (j+1) +
                                     ".Idle")>=0) {
                            taskRun[j]+="0";
                        }
                        else {
                            taskRun[j]+="x";
                        }
                    }
                }
            }
            if (runningTaskTemp > 0 && runningTask==0) {
                runningTask = runningTaskTemp;
            }
        }
    }
    else {
        runningTask = 0;
        if (!(strLine.indexOf(".Running")>0
              || strLine.indexOf(".Idle ")>0
              || strLine.indexOf(".Start ")>0
              || strLine.indexOf(".Ready ")>0
              || strLine.indexOf(".MissedDeadline")>0
              || boolCountIdleTasks)
```

```
                && !allFinish(strLine)) {
                boolCountIdleTasks = true;
                int totalClock = 0;

                if(!taskSingle[0].offsetExceeded)
                    timeBeforeNextRun = taskSingle[0].o-clock[0];
                else
                    timeBeforeNextRun = taskSingle[0].p-clock[0];


                for (int i = 1; i < tn; i++) {
                    if (!taskSingle[i].offsetExceeded) {
                        if (timeBeforeNextRun > taskSingle[i].o-clock[i]) {
                            timeBeforeNextRun = taskSingle[i].o-clock[i];
                        }
                    }
                    else {
                        if (!taskSingle[i].nonperiodic &&
                            timeBeforeNextRun > (taskSingle[i].p-clock[i])) {
                            timeBeforeNextRun = taskSingle[i].p-clock[i];
                        }
                    }
                }
                for (int j = 0; j<tn; j++) {
                    for (int i = 0; i<timeBeforeNextRun; i++) {
                        if (!taskSingle[j].offsetExceeded) {
                            taskRun[j] += "-";
                        }
                        else if (strLine.indexOf("Taskc" + (j+1) + ".Finish")>0) {
                            taskRun[j] += "*";
                        }
                        else {
                            taskRun[j] += "0";
                        }

                    }
                    clock[j] += timeBeforeNextRun;
                }
            }
        }
    }

   /**
    * Prints the schedules created in the Parser-class.
    * The tasks is printed according to the way they were created.
    */
    public void printTrace() {
        String[] temp = new String[tn];
        String line = "";
        int iterator = 0;
        while (iterator<tn) {
            line = "";
            for (int i = 0; i<tn; i++) {
                if (taskSingle[i].taskID == (iterator+1)) {
                    line = taskRun[i];
                    break;
                }
            }
            temp[iterator++] = line;
        }

        System.out.print("\t");
        for (int i = 1; i<=taskRun[0].length(); i++) {
            if (i < 9 && (i%5)==0) {
                System.out.print(i);
            }
            else if(i>=9 && i<=97 && ((i+1)%5)==0) {
                System.out.print((i+1));
            }
            else if ((i>=98 && (i+2)%5==0)) {
                System.out.print((i+2));
            }
            else if((i>=9 && (i%5)==0) || (i>=99 && (i%5==0 || i%5==4))) {

            }
            else {
                System.out.print(" ");
            }
        }
        System.out.print("\n");
        for(int i = 0; i<taskRun.length; i++) {
            if (!taskSingle[i].limitTask)
                System.out.println("Task: " + (i+1) + "\t" + temp[i]);
            //System.out.println("clocks:\t" + clocks[i] + "\n\n");
        }
    }
```

```
    /**
     * Prints an explanation of a schedule
     */
     public void printExplain() {
         System.out.println("\n1 = running\n0 = idle\n- = offset\nx = Missed
              deadline\nX = Missed deadline running\n* = finished");
     }

    /**
     * Prints the scheduling algorithms used on the different processors.
     * @param ps Processor[] containing the processors of the system
     */
     public void printSched(Processor[] ps) {
       for (int l = 0; l<ps.length; l++) {
         if (l>0) {
           System.out.print(", ");
         }
         switch(ps[l].schedType) {
           case 1:
             System.out.print("EDF");
             break;
           case 2:
             System.out.print("RM");
             break;
           case 3:
             System.out.print("DM");
             break;
           case 4:
             System.out.print("FP");
             break;
         }
       }
       System.out.print("\n");
     }

    /**
     * Print tasks writes the mapping of tasks to the command-prompt.
     * @param ts Task[] containing all tasks in the system.
     */
     public void printTasks(Task[] ts) {
         for (int i=0; i<ts.length; i++) {
             System.out.println("Task: " + ts[i].pri + " on proc: " + ts[i].proc);
         }
     }

}
```

E.9 Platform.java

```
/**
* Platform.java
* Platform class for MoVES.
* The Platform class contains information about the
* platform in the constructed MPSoC system.
* The Platform is created with a number of processors.
*
* The Platform-class handles processor-speed and calculation of hyperperiod.
*
* @author    Kristian Staaloe Knudsen, Jens Ellebaek Nielsen
* @version   1.1
* @access    public
*/
package MoVES;

import java.text.*;

public class Platform{
     /** All processors in the system */
     public Processor[] processors;
     /** Utilisation array */
     public double[] utilArr;
     /** Utilisation array */
     public double[] uar;

    /**
     * Constructor
     * @param ps Processor[] containing all processors in the system.
     */
     public Platform(Processor[] ps){
         processors=ps;
         utilArr = new double[processors.length];
```

```java
        uar = new double[processors.length];
        for (int i = 0; i<processors.length;i++) {
            utilArr[i]=0;
            uar[i] = 0;
        }
    }

    /**
     * Returns a integer[] containing all different processor speeds in the system.
     * @return int[] Integerarray with processor-speeds.
     */
    public int[] procSpeed() {
        int[] temp = new int[processors.length];
        for (int i = 0; i < processors.length; i++) temp[i] =
            processors[i].frequency;
        return temp;
    }

    /**
     * Calculate the hyper period of ann tasks in the system.
     * @param ta Task[] containing all tasks in the system.
     * @return int Integer containing the hyperperiod of the system.
     */
    public int calcHyperPeriod(Task[] ta) {
        int[] tempPeriods = new int[ta.length];
        int maxOffset = 0;
        int iterator = 0;
        tempPeriods[0] = ta[0].p;
        maxOffset = ta[0].o;
        for (int i = 1; i<ta.length; i++) {
            if (!ta[i].nonperiodic) {
                tempPeriods[++iterator] = ta[i].p;
                if (ta[i].o>maxOffset)
                    maxOffset = ta[i].o;
            }
        }
        int[] periods = new int[++iterator];
        for (int i = 0; i<iterator; i++)
            periods[i] = tempPeriods[i];
        int temp = (int)(calcLCM(periods, 0, 1)+maxOffset);
        return temp;
    }

    /**
     * Calculate LCM of a integer[]
     * Used to calculate LCM of the processor-speed
     * @param sp Integer[] containing the processor-speeds
     * @param iterator Integer of how long into the array have been calculated.
     * @param tempLCM double containing the current LCM
     * @return int Containing the LCM.
     */
    public int calcLCM(int[] sp, int iterator, double tempLcm) {
        if (iterator == sp.length-1) {
            return ((int)lcmAux(sp[iterator], tempLcm));
        }
        else {
            return calcLCM(sp, iterator+1, lcmAux(sp[iterator], tempLcm));
        }
    }

    /**
     * Aux function to calculate LCM.
     * @param a and b
     */
    public double lcmAux(double a, double b) {
        return (a*b)/gcd((int)a,(int)b);
    }

    /**
     * calculate GCD
     * @return int Containing the GCD
     */
    public int gcd(int a, int b) {
        if (b == 0) return a;
        else return gcd(b, a % b);
    }

    /**
     * Find the least value of an integer[]
     * @param sp An integer[]
     * @return int containing the least value of the array.
     */
    public int leastValue(int[] sp) {
        int min = sp[0];
        for (int i =1; i<sp.length;i++) if (sp[i]<min) min = sp[i];
        return min;
    }
```

```
    /**
     * Calculate the utilisation of each processor.
     * @param ta Task[] containing the tasks in the system.
     */
    public void calcUtilisation(Task[] ta) {
        for (int i = 0; i<utilArr.length; i++) {
            uar[i] = 0;
            utilArr[i] = 0;
        }

        for (int i = 0; i<ta.length; i++) {
            if (!ta[i].nonperiodic) {
                utilArr[ta[i].proc-1] += (double)ta[i].emax/(double)ta[i].p;
                if (!(ta[i].sd > 0)) {
                    uar[ta[i].proc-1] += (double)ta[i].emax/(double)ta[i].p;
                }
            }
        }
    }

    /**
     * check the utilisation.
     * If it is above 1.00 then the system cannot be scheduled.
     * @return boolean utilisation <1.00
     */
    public boolean checkUtil() {
        for (int i = 0; i<uar.length; i++)
            if (uar[i] > 1.00)
                return false;
        return true;
    }

    /**
     * Print the utilisation
     * Writes the utilisation from each processor to the console.
     */
    public void printUtilisation() {
        DecimalFormat format_number = new DecimalFormat("#0.00");
        System.out.println("Utilization:");

        for (int i = 0; i<utilArr.length;i++)
            System.out.println("Processor " + (i+1) + ":\t" +
                format_number.format(utilArr[i]));

    }

}
```

E.10 Resource.java

```
/**
* Resource.java
* Resource class for MoVES.
* The Resource class handles the resources defined in MPSoC.java.
*
* A resource can be used to make a task non-preemptable.
*
* @author    Kristian Staaloe Knudsen, Jens Ellebaek Nielsen
* @version   1.1
* @access    public
*/
package MoVES;

public class Resource{
    /** Static resource counter */
    public static int resourceCount=0;
    /** Id for the resource object */
    public int id;
    /** All resources in the system */
    public static Resource[] rs = new Resource[1];

    /**
     * Constructor
     */
    public Resource(){
        id = ++resourceCount;
        Resource[] temp = new Resource[resourceCount];
        for (int i = 0; i<resourceCount-1; i++) {
            temp[i] = rs[i];
        }
```

```java
        temp[resourceCount-1] = this;
        rs = new Resource[resourceCount];
        for (int i = 0; i<resourceCount; i++)
            rs[i] = temp[i];
    }

    /**
     * Empty constructor to get global variables from the resource class
     * @param build Boolean to ensure the empty constructor is desided
     */
    public Resource(boolean build) {
        //empty constructor
    }

    /**
     * Get the resource array
     * @return Resource[] Containing all resources defined in the system.
     */
    public Resource[] getResources() {
        return rs;
    }

    /**
     * Reset the resource object
     * reset the static counter.
     */
    public void reset() {
        rs = new Resource[1];
        resourceCount = 0;
    }
}
```

E.11 Cost.java

```java
/**
* Cost.java
* Cost class for MoVES.
* The Cost class contains information about the
* Costs in the constructed MPSoC system from MPSoC.java
*
* A cost is constructed using a Task[][] containing a mapping
* of tasks onto processing elements.
*
* @author    Kristian Staaloe Knudsen, Jens Ellebaek Nielsen
* @version   1.1
* @access    public
*/
package MoVES;

class Cost {
    /** The costs for all tasks in the system */
    public int[][] costs;
    /** All tasks defined in the system. */
    public Task[][] ta;

    /** Definition of shared costs in the system */
    public int[][] shared;

    /**
     * Constructor
     * @param ta Task[][] containing a mapping of tasks to processors.
     */
    public Cost(Task[][] ta) {
        int nrTasks = 0;
        this.ta = ta;
        for (int i = 0; i<ta.length; i++)
            nrTasks+=ta[i].length;
        costs = new int[nrTasks][5];
        shared = new int[nrTasks][nrTasks];
        for (int i = 0; i<costs.length;i++) {
            for (int j = 0; j<costs[i].length;j++)
                costs[i][j] = 0;
            for (int k = 0; k<nrTasks; k++)
                shared[i][k] = 0;
        }

    }

    /**
     * Sets a task to use the desired costs.
     * @param t Task which uses the cost
```

```
    * @param costStatic, costIdle, costReady, costRunning, costPreempt all of types
         int.
    */
    public void set(Task t, int costStatic, int costIdle, int costReady, int
         costRunning, int costPreempt) {
        int nr = 0;
        for (int i = 0; i<ta.length;i++)
            for (int j = 0; j<ta[i].length; j++)
                if (ta[i][j].taskID == t.taskID) {
                    costs[nr][0] = costStatic;
                    costs[nr][1] = costIdle;
                    costs[nr][2] = costReady;
                    costs[nr][3] = costRunning;
                    costs[nr][4] = costPreempt;
                }
                else
                    nr++;
    }

   /**
    * Share a cost (e.g. t1 writes 5 bytes of data to t2)
    * @param from Task who writes the data
    * @param to Task the data is written for
    * @param amount How must data is written of type int.
    */
    public void share(Task from, Task to, int amount) {
        int intFrom=0, intTo=0;
        int nr = 0;
        for (int i = 0; i<ta.length; i++) {
            for (int j = 0; j<ta[i].length; j++) {
                if (ta[i][j].taskID == from.taskID)
                    intFrom = nr;
                if (ta[i][j].taskID == to.taskID)
                    intTo = nr;
                nr++;
            }
        }
        shared[intFrom][intTo] = amount;
    }
}
```

E.12 help.txt

```
Usage: javafrontend [-options]
           to execute a MPSoC-system specified in the file system.java
           checking the property speciefied in bool.q

where options include:
    -s [scheduling algorithms...] To verify the MPSoC-system with
                                  the specified scheduling algorithms
                                  {EDF, RM, FP, DM}

    -a               To verify all possible mappings of tasks on the
                     processors specified in the MPSoC system.

    -t[limit]        Writes the schedule for the MPSoC-system until
                     property satisfied OR the specified limit.
                     If limit == 0 or not specified limit will be chosen to
                     the hyperperiod

    -g <granularity> describes a granularity which will be used when constructing
         the system.

    -u               Writes the utilasation for each processing element

    -o <filename>    Creates the Uppaal-model in the file specified in filename

    -f <queryfile>   Specifies which file contains the queries for current
         verification
                     (default: bool.q)

    -e               Writes the notation of the trace to the screen, after the trace
         has been proceeded

    -h -? -help      Prints this help message.
```

E.13 MPSoC.java for Smart Phone

```java
public class MPSoC {
    public Application apps;
    public Platform pl;

    public MPSoC(int granularity) {
        //Definitions of task-types.. Can be exchanged with other arrays, according
            to desired platform. Currently GPP.
        int[] ttype = {2882, 200, 100, 981, 395, 2952, 2000, 160, 115, 552, 15200,
             15596, 2882, 3617, 23628, 85864, 3460, 472, 315, 157, 52, 197, 26154,
             3617, 26007, 5013, 5014, 1140, 1966, 787, 472, 200, 3617, 1500, 1200,
             41580, 138666, 180246, 177474, 122034, 36781, 14172, 38982, 144924,
             79250, 135890, 1071, 476, 63914, 2568, 21305, 266687, 78462, 1060, 123,
              55, 471, 16370, 140310, 9154};

        //Other task declarations. Rename to ttype to use these declarations.
        int[] asic0 = {222, 142, 40, 125, 146, 42, 232, 120, 32, 41, 652, 868, 293,
             221, 221, 178, 765, 979, 132, 142, 116, 128, 122, 453, 192, 759, 267,
             329, 114, 196, 78, 47, 120, 136, 115, 220, 643, 1270, 1809, 1980, 1220,
              620, 236, 971, 2371, 1448, 2112, 582, 146, 1282, 4324, 639, 5439, 862,
              60, 113, 155, 72, 175, 1690, 203};
        int[] asic1 = {223, 162, 73, 228, 148, 63, 345, 178, 72, 63, 867, 1438, 328,
              123, 438, 542, 1265, 245, 356, 191, 342, 121, 527, 267, 728, 189, 561,
              544, 259, 87, 163, 353, 178, 190, 182, 1872, 1152, 1982, 2621, 1792,
             901, 515, 874, 2538, 1983, 3871, 773, 541, 1826, 3992, 501, 2176, 834,
             87, 164, 182, 98, 289, 1469, 323};
        int[] FPGA0 = {288, 160, 120, 163, 89, 298, 240, 120, 153, 152, 4547, 1519,
             288, 361, 2362, 8586, 346, 172, 115, 65, 67, 193, 2615, 661, 2600, 601,
              701, 314, 296, 143, 222, 98, 361, 250, 220, 4158, 13866, 18024, 17747,
              12203, 3678, 1417, 3898, 14492, 7925, 13589, 407, 276, 9178, 25755,
             6391, 22668, 1853, 687, 1023, 312, 1091, 932, 21469, 1323};
        //Define the bus-resource
        Resource bus = new Resource();

        //Defines tasks (Execution time, Deadline, offset, period, FP)
        //Gsm Decoder
        Task t0_0  = new Task(ttype[0], ttype[0], 0.02, 0, 0.02, 1);
        Task t0_1  = new Task(ttype[1], ttype[1], 0.02, 0, 0.02, 2);
        Task t0_2  = new Task(ttype[1], ttype[1], 0.02, 0, 0.02, 3);
        Task t0_3  = new Task(ttype[1], ttype[1], 0.02, 0, 0.02, 4);
        Task t0_4  = new Task(ttype[1], ttype[1], 0.02, 0, 0.02, 5);
        Task t0_5  = new Task(ttype[2], ttype[2], 0.02, 0, 0.02, 6);
        Task t0_6  = new Task(ttype[3], ttype[3], 0.02, 0, 0.02, 7);
        Task t0_7  = new Task(ttype[4], ttype[4], 0.02, 0, 0.02, 8);
        Task t0_8  = new Task(ttype[5], ttype[5], 0.02, 0, 0.02, 9);
        Task t0_9  = new Task(ttype[3], ttype[3], 0.02, 0, 0.02, 10);
        Task t0_10 = new Task(ttype[4], ttype[4], 0.02, 0, 0.02, 11);
        Task t0_11 = new Task(ttype[5], ttype[5], 0.02, 0, 0.02, 12);
        Task t0_12 = new Task(ttype[3], ttype[3], 0.02, 0, 0.02, 13);
        Task t0_13 = new Task(ttype[4], ttype[4], 0.02, 0, 0.02, 14);
        Task t0_14 = new Task(ttype[5], ttype[5], 0.02, 0, 0.02, 15);
        Task t0_15 = new Task(ttype[3], ttype[3], 0.02, 0, 0.02, 16);
        Task t0_16 = new Task(ttype[4], ttype[4], 0.02, 0, 0.02, 17);
        Task t0_17 = new Task(ttype[5], ttype[5], 0.02, 0, 0.02, 18);
        Task t0_18 = new Task(ttype[6], ttype[6], 0.02, 0, 0.02, 19);
        Task t0_19 = new Task(ttype[7], ttype[7], 0.02, 0, 0.02, 20);
        Task t0_20 = new Task(ttype[12], ttype[12], 0.02, 0, 0.02, 21);
        Task t0_21 = new Task(ttype[8], ttype[8], 0.02, 0, 0.02, 22);
        Task t0_22 = new Task(ttype[9], ttype[9], 0.02, 0, 0.02, 23);
        Task t0_23 = new Task(ttype[10], ttype[10], 0.02, 0, 0.02, 24);
        Task t0_24 = new Task(ttype[54], ttype[54], 0.02, 0, 0.02, 25);
        Task t0_25 = new Task(ttype[9], ttype[9], 0.02, 0, 0.02, 26);
        Task t0_26 = new Task(ttype[57], ttype[57], 0.02, 0, 0.02, 27);
        Task t0_27 = new Task(ttype[8], ttype[8], 0.02, 0, 0.02, 28);
        Task t0_28 = new Task(ttype[9], ttype[9], 0.02, 0, 0.02, 29);
        Task t0_29 = new Task(ttype[10], ttype[10], 0.02, 0, 0.02, 30);
        Task t0_30 = new Task(ttype[53], ttype[53], 0.02, 0, 0.02, 31);
        Task t0_31 = new Task(ttype[9], ttype[9], 0.02, 0, 0.02, 32);
        Task t0_32 = new Task(ttype[58], ttype[58], 0.02, 0, 0.02, 33);
        Task t0_33 = new Task(ttype[11], ttype[11], 0.02, 0, 0.02, 34);

//GSM Encoder
        Task t1_0  = new Task(ttype[13], ttype[13], 0.02, 0, 0.02, 1);
        Task t1_1  = new Task(ttype[14], ttype[14], 0.02, 0, 0.02, 1);
        Task t1_2  = new Task(ttype[15], ttype[15], 0.02, 0, 0.02, 1);
        Task t1_3  = new Task(ttype[16], ttype[16], 0.02, 0, 0.02, 1);
        Task t1_4  = new Task(ttype[17], ttype[17], 0.02, 0, 0.02, 1);
        Task t1_5  = new Task(ttype[18], ttype[18], 0.02, 0, 0.02, 1);
        Task t1_6  = new Task(ttype[7], ttype[7], 0.02, 0, 0.02, 1);
        Task t1_7  = new Task(ttype[20], ttype[20], 0.02, 0, 0.02, 1);
        Task t1_8  = new Task(ttype[9], ttype[9], 0.02, 0, 0.02, 1);
        Task t1_9  = new Task(ttype[22], ttype[22], 0.02, 0, 0.02, 1);
        Task t1_10 = new Task(ttype[55], ttype[55], 0.02, 0, 0.02, 1);
        Task t1_11 = new Task(ttype[9], ttype[9], 0.02, 0, 0.02, 1);
```

```
        Task t1_12 = new Task(ttype[59], ttype[59], 0.02, 0, 0.02, 1);
        Task t1_13 = new Task(ttype[20], ttype[20], 0.02, 0, 0.02, 1);
        Task t1_14 = new Task(ttype[9], ttype[9], 0.02, 0, 0.02, 1);
        Task t1_15 = new Task(ttype[22], ttype[22], 0.02, 0, 0.02, 1);
        Task t1_16 = new Task(ttype[56], ttype[56], 0.02, 0, 0.02, 1);
        Task t1_17 = new Task(ttype[9], ttype[9], 0.02, 0, 0.02, 1);
        Task t1_18 = new Task(ttype[52], ttype[52], 0.02, 0, 0.02, 1);
        Task t1_19 = new Task(ttype[23], ttype[23], 0.02, 0, 0.02, 1);
        Task t1_20 = new Task(ttype[24], ttype[24], 0.02, 0, 0.02, 1);
        Task t1_21 = new Task(ttype[25], ttype[25], 0.02, 0, 0.02, 1);
        Task t1_22 = new Task(ttype[26], ttype[26], 0.02, 0, 0.02, 1);
        Task t1_23 = new Task(ttype[27], ttype[27], 0.02, 0, 0.02, 1);
        Task t1_24 = new Task(ttype[28], ttype[28], 0.02, 0, 0.02, 1);
        Task t1_25 = new Task(ttype[3], ttype[3], 0.02, 0, 0.02, 1);
        Task t1_26 = new Task(ttype[4], ttype[4], 0.02, 0, 0.02, 1);
        Task t1_27 = new Task(ttype[31], ttype[31], 0.02, 0, 0.02, 1);
        Task t1_28 = new Task(ttype[24], ttype[24], 0.02, 0, 0.02, 1);
        Task t1_29 = new Task(ttype[25], ttype[25], 0.02, 0, 0.02, 1);
        Task t1_30 = new Task(ttype[26], ttype[26], 0.02, 0, 0.02, 1);
        Task t1_31 = new Task(ttype[27], ttype[27], 0.02, 0, 0.02, 1);
        Task t1_32 = new Task(ttype[28], ttype[28], 0.02, 0, 0.02, 1);
        Task t1_33 = new Task(ttype[3], ttype[3], 0.02, 0, 0.02, 1);
        Task t1_34 = new Task(ttype[4], ttype[4], 0.02, 0, 0.02, 1);
        Task t1_35 = new Task(ttype[31], ttype[31], 0.02, 0, 0.02, 1);
        Task t1_36 = new Task(ttype[24], ttype[24], 0.02, 0, 0.02, 1);
        Task t1_37 = new Task(ttype[25], ttype[25], 0.02, 0, 0.02, 1);
        Task t1_38 = new Task(ttype[26], ttype[26], 0.02, 0, 0.02, 1);
        Task t1_39 = new Task(ttype[27], ttype[27], 0.02, 0, 0.02, 1);
        Task t1_40 = new Task(ttype[28], ttype[28], 0.02, 0, 0.02, 1);
        Task t1_41 = new Task(ttype[3], ttype[3], 0.02, 0, 0.02, 1);
        Task t1_42 = new Task(ttype[4], ttype[4], 0.02, 0, 0.02, 1);
        Task t1_43 = new Task(ttype[31], ttype[31], 0.02, 0, 0.02, 1);
        Task t1_44 = new Task(ttype[24], ttype[24], 0.02, 0, 0.02, 1);
        Task t1_45 = new Task(ttype[25], ttype[25], 0.02, 0, 0.02, 1);
        Task t1_46 = new Task(ttype[26], ttype[26], 0.02, 0, 0.02, 1);
        Task t1_47 = new Task(ttype[27], ttype[27], 0.02, 0, 0.02, 1);
        Task t1_48 = new Task(ttype[28], ttype[28], 0.02, 0, 0.02, 1);
        Task t1_49 = new Task(ttype[3], ttype[3], 0.02, 0, 0.02, 1);
        Task t1_50 = new Task(ttype[4], ttype[4], 0.02, 0, 0.02, 1);
        Task t1_51 = new Task(ttype[31], ttype[31], 0.02, 0, 0.02, 1);
        Task t1_52 = new Task(ttype[32], ttype[32], 0.02, 0, 0.02, 1);
//JPEG Decoder
        Task t2_0  = new Task(ttype[40], ttype[40], 0.5, 0, 0.5, 1);
        Task t2_1  = new Task(ttype[41], ttype[41], 0.5, 0, 0.5, 1);
        Task t2_2  = new Task(ttype[42], ttype[42], 0.5, 0, 0.5, 1);
        Task t2_3  = new Task(ttype[43], ttype[43], 0.5, 0, 0.5, 1);
        Task t2_4  = new Task(ttype[44], ttype[44], 0.5, 0, 0.5, 1);
        Task t2_5  = new Task(ttype[45], ttype[45], 0.5, 0, 0.5, 1);

//MP3 Decoder
        Task t3_0  = new Task(ttype[46], ttype[46], 0.025, 0, 0.025, 1);
        Task t3_1  = new Task(ttype[47], ttype[47], 0.025, 0, 0.025, 2);
        Task t3_2  = new Task(ttype[47], ttype[47], 0.025, 0, 0.025, 3);
        Task t3_3  = new Task(ttype[40], ttype[40], 0.025, 0, 0.025, 4);
        Task t3_4  = new Task(ttype[40], ttype[40], 0.025, 0, 0.025, 5);
        Task t3_5  = new Task(ttype[41], ttype[41], 0.025, 0, 0.025, 1);
        Task t3_6  = new Task(ttype[41], ttype[41], 0.025, 0, 0.025, 6);
        Task t3_7  = new Task(ttype[48], ttype[48], 0.025, 0, 0.025, 7);
        Task t3_8  = new Task(ttype[49], ttype[49], 0.025, 0, 0.025, 8);
        Task t3_9  = new Task(ttype[49], ttype[49], 0.025, 0, 0.025, 9);
        Task t3_10 = new Task(ttype[50], ttype[50], 0.025, 0, 0.025, 10);
        Task t3_11 = new Task(ttype[50], ttype[50], 0.025, 0, 0.025, 5);
        Task t3_12 = new Task(ttype[43], ttype[43], 0.025, 0, 0.025, 6);
        Task t3_13 = new Task(ttype[43], ttype[43], 0.025, 0, 0.025, 7);
        Task t3_14 = new Task(ttype[51], ttype[51], 0.025, 0, 0.025, 8);
        Task t3_15 = new Task(ttype[51], ttype[51], 0.025, 0, 0.025, 9);
        Task t3_m1 = new Task(1, 1, 0.025, 0, 0.025, 10);
        Task t3_m2 = new Task(1, 1, 0.025, 0, 0.025, 10);
        Task t3_m3 = new Task(1, 1, 0.025, 0, 0.025, 10);

//JPEG Encoder
        Task t4_0  = new Task(ttype[35], ttype[35], 0.25, 0, 0.25, 1);
        Task t4_1  = new Task(ttype[36], ttype[36], 0.25, 0, 0.25, 1);
        Task t4_2  = new Task(ttype[37], ttype[37], 0.25, 0, 0.25, 1);
        Task t4_3  = new Task(ttype[38], ttype[38], 0.25, 0, 0.25, 1);
        Task t4_4  = new Task(ttype[39], ttype[39], 0.25, 0, 0.25, 1);


        //Defines processors
        Processor p0_1 = new Processor(25000000, Processor.RM, Processor.NPCS);
        Processor p0_2 = new Processor(25000000, Processor.RM, Processor.NPCS);

        Processor p1_1 = new Processor(25000000, Processor.RM, Processor.NPCS);

        Processor p3_1 = new Processor(25000000, Processor.RM, Processor.NPCS);
        Processor p3_2 = new Processor(25000000, Processor.RM, Processor.NPCS);
        Processor p3_m = new Processor(25000000, Processor.RM, Processor.NPCS);
```

```
        Processor p4_1 = new Processor(25000000,Processor.RM, Processor.NPCS);


        //Assigns tasks to processors
Task[][] tasks = {{t0_0, t0_1, t0_2, t0_3, t0_4, t0_5, t0_6, t0_7, t0_8, t0_9,
    t0_10, t0_11, t0_12, t0_13, t0_14, t0_15, t0_16, t0_17, t0_18, t0_19, t0_20,
    t0_21, t0_22, t0_23, t0_24, t0_25, t0_26, t0_27, t0_28, t0_29, t0_30, t0_31},
                        {t0_32, t0_33},
                        {t1_0, t1_1, t1_2, t1_3, t1_4, t1_5, t1_6, t1_7, t1_8,
                             t1_9, t1_10, t1_11, t1_12, t1_13, t1_14, t1_15,
                             t1_16, t1_17, t1_18, t1_19, t1_20, t1_21, t1_22,
                             t1_23, t1_24, t1_25, t1_26, t1_27, t1_28, t1_29,
                             t1_30, t1_31, t1_32, t1_33, t1_34, t1_35, t1_36,
                             t1_37, t1_38, t1_39, t1_40, t1_41, t1_42, t1_43,
                             t1_44, t1_45, t1_46, t1_47, t1_48, t1_49, t1_50,
                             t1_51, t1_52},
                        {t3_0, t3_1, t3_3, t3_5, t3_9, t3_11, t3_13, t3_15},
                        {t3_2, t3_4, t3_6, t3_7, t3_8, t3_10, t3_12, t3_14},
                        {t3_m1, t3_m2, t3_m3},
                        {t2_0, t2_1, t2_3, t2_4, t2_5, t4_0, t4_1, t4_2, t4_3,
                             t4_4}};

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

        //Adds the processors to the system
        Processor[] ps = {p0_1,p0_2,p1_1,p3_1,p3_2,p3_m, p4_1};

        pl = new Platform(ps);
        apps = new Application(tasks, ca, granularity);

        apps.useResource(t3_m1,bus);
        apps.useResource(t3_m2,bus);
        apps.useResource(t3_m3,bus);

        //Adds dependencies to the system.
        //The dependencies references to the name of the task-object
        apps.addDep(t0_0,t0_8);
        apps.addDep(t0_0,t0_21);
        apps.addDep(t0_0,t0_24);
        apps.addDep(t0_0,t0_27);
        apps.addDep(t0_0,t0_30);
        apps.addDep(t0_0,t0_33);
        apps.addDep(t0_1,t0_6);
        apps.addDep(t0_1,t0_7);
        apps.addDep(t0_1,t0_8);
        apps.addDep(t0_2,t0_9);
        apps.addDep(t0_2,t0_10);
        apps.addDep(t0_2,t0_11);
        apps.addDep(t0_3,t0_12);
        apps.addDep(t0_3,t0_13);
        apps.addDep(t0_3,t0_14);
        apps.addDep(t0_4,t0_15);
        apps.addDep(t0_4,t0_16);
        apps.addDep(t0_4,t0_17);
        apps.addDep(t0_5,t0_19);
        apps.addDep(t0_6,t0_7);
        apps.addDep(t0_7,t0_8);
        apps.addDep(t0_8,t0_11);
        apps.addDep(t0_9,t0_10);
        apps.addDep(t0_10,t0_11);
        apps.addDep(t0_11,t0_14);
        apps.addDep(t0_12,t0_13);
        apps.addDep(t0_13,t0_14);
        apps.addDep(t0_14,t0_17);
        apps.addDep(t0_15,t0_16);
        apps.addDep(t0_16,t0_17);
        apps.addDep(t0_17,t0_18);
        apps.addDep(t0_17,t0_20);
        apps.addDep(t0_18,t0_23);
        apps.addDep(t0_18,t0_26);
        apps.addDep(t0_18,t0_29);
        apps.addDep(t0_18,t0_32);
        apps.addDep(t0_19,t0_21);
        apps.addDep(t0_19,t0_24);
        apps.addDep(t0_19,t0_27);
        apps.addDep(t0_19,t0_30);
        apps.addDep(t0_21,t0_22);
        apps.addDep(t0_22,t0_23);
        apps.addDep(t0_23,t0_26);
        apps.addDep(t0_23,t0_33);
        apps.addDep(t0_24,t0_25);
        apps.addDep(t0_25,t0_26);
        apps.addDep(t0_26,t0_29);
        apps.addDep(t0_26,t0_33);
        apps.addDep(t0_27,t0_28);
```

```
apps.addDep(t0_28,t0_29);
apps.addDep(t0_29,t0_32);
apps.addDep(t0_29,t0_33);
apps.addDep(t0_30,t0_31);
apps.addDep(t0_31,t0_32);
apps.addDep(t0_32,t0_20);
apps.addDep(t0_32,t0_33);

apps.addDep(t1_0,t1_1);
apps.addDep(t1_0,t1_7);
apps.addDep(t1_0,t1_10);
apps.addDep(t1_0,t1_13);
apps.addDep(t1_0,t1_16);
apps.addDep(t1_0,t1_9);
apps.addDep(t1_0,t1_12);
apps.addDep(t1_0,t1_15);
apps.addDep(t1_0,t1_18);
apps.addDep(t1_1,t1_2);
apps.addDep(t1_1,t1_9);
apps.addDep(t1_1,t1_20);
apps.addDep(t1_1,t1_52);
apps.addDep(t1_2,t1_3);
apps.addDep(t1_3,t1_4);
apps.addDep(t1_4,t1_5);
apps.addDep(t1_5,t1_6);
apps.addDep(t1_6,t1_7);
apps.addDep(t1_6,t1_10);
apps.addDep(t1_6,t1_13);
apps.addDep(t1_6,t1_16);
apps.addDep(t1_7,t1_8);
apps.addDep(t1_8,t1_9);
apps.addDep(t1_9,t1_12);
apps.addDep(t1_9,t1_19);
apps.addDep(t1_10,t1_11);
apps.addDep(t1_11,t1_12);
apps.addDep(t1_12,t1_15);
apps.addDep(t1_12,t1_19);
apps.addDep(t1_13,t1_14);
apps.addDep(t1_14,t1_15);
apps.addDep(t1_15,t1_18);
apps.addDep(t1_15,t1_19);
apps.addDep(t1_16,t1_17);
apps.addDep(t1_17,t1_18);
apps.addDep(t1_18,t1_19);
apps.addDep(t1_18,t1_52);
apps.addDep(t1_19,t1_20);
apps.addDep(t1_19,t1_28);
apps.addDep(t1_19,t1_36);
apps.addDep(t1_19,t1_44);
apps.addDep(t1_20,t1_21);
apps.addDep(t1_21,t1_22);
apps.addDep(t1_22,t1_23);
apps.addDep(t1_23,t1_24);
apps.addDep(t1_23,t1_26);
apps.addDep(t1_24,t1_25);
apps.addDep(t1_25,t1_26);
apps.addDep(t1_26,t1_27);
apps.addDep(t1_27,t1_28);
apps.addDep(t1_28,t1_29);
apps.addDep(t1_29,t1_30);
apps.addDep(t1_29,t1_35);
apps.addDep(t1_30,t1_31);
apps.addDep(t1_31,t1_32);
apps.addDep(t1_31,t1_34);
apps.addDep(t1_32,t1_33);
apps.addDep(t1_33,t1_34);
apps.addDep(t1_34,t1_35);
apps.addDep(t1_35,t1_36);
apps.addDep(t1_36,t1_37);
apps.addDep(t1_37,t1_38);
apps.addDep(t1_37,t1_43);
apps.addDep(t1_38,t1_39);
apps.addDep(t1_39,t1_40);
apps.addDep(t1_39,t1_42);
apps.addDep(t1_40,t1_41);
apps.addDep(t1_41,t1_42);
apps.addDep(t1_42,t1_43);
apps.addDep(t1_43,t1_44);
apps.addDep(t1_44,t1_45);
apps.addDep(t1_45,t1_46);
apps.addDep(t1_46,t1_47);
apps.addDep(t1_47,t1_48);
apps.addDep(t1_47,t1_50);
apps.addDep(t1_48,t1_49);
apps.addDep(t1_49,t1_50);
apps.addDep(t1_50,t1_51);
apps.addDep(t1_51,t1_52);
```

```
        apps.addDep(t2_0, t2_1);
        apps.addDep(t2_1, t2_2);
        apps.addDep(t2_2, t2_3);
        apps.addDep(t2_3, t2_4);
        apps.addDep(t2_4, t2_5);

        apps.addDep(t3_0,t3_m1);
        apps.addDep(t3_m1,t3_2);
        apps.addDep(t3_2,t3_4);
        apps.addDep(t3_4,t3_6);
        apps.addDep(t3_6,t3_7);
        apps.addDep(t3_7,t3_8);
        apps.addDep(t3_8,t3_10);
        apps.addDep(t3_10,t3_12);
        apps.addDep(t3_12,t3_14);
        apps.addDep(t3_0,t3_1);
        apps.addDep(t3_1,t3_3);
        apps.addDep(t3_3,t3_5);
        apps.addDep(t3_5,t3_m2);
        apps.addDep(t3_m2,t3_7);
        apps.addDep(t3_7,t3_m3);
        apps.addDep(t3_m3,t3_9);
        apps.addDep(t3_9,t3_11);
        apps.addDep(t3_11,t3_13);
        apps.addDep(t3_13,t3_15);

        apps.addDep(t4_0,t4_1);
        apps.addDep(t4_1,t4_2);
        apps.addDep(t4_2,t4_3);
        apps.addDep(t4_3,t4_4);

    }
}
```

Appendix F

Entire Uppaal model

This appendix contains model, arguments and local declarations for all templates in the UPPAAL model. The system generated consists of two processors with three tasks in all.

F.1 Global declarations

```
const int M = 2; //The number of Processors
const int N = 2; //The maximum number of tasks per Processor
const int MN = 3; //The total number of tasks
const int R = 1;//The total number of resources
const int NOC = 2; //Total number of costs

//symbolic representation of scheduling actions
const int REA = 0, RUN = 1, PRE = 2, FIN = 3;

//symbolic representation of internal processor synchronizing
const int SYN = 0, SCH = 1, ALL = 2;

//symbolic representation of states for cost-model
const int STATIC = 0, IDLE = 1, READY = 2, RUNNING = 3, PREEMPTED=4;
//Symbolic representation of memory and power.
const int Memory = 0, Power = 1;

//Symbolic representation of cost types.
const int HARD = 0, SOFT = 1, FIRM = 2;

//symbolic representation of scheduling principles
const int FP = 0, RM = 1, DM = 2, EDF = 3;

//symbolic representation of allocation algorithms
const int PCS=0, PRI_INH = 1, NPCS = 2;

int[0,3] processorScheduling[M] = {RM, FP};
//Contains information about the scheduling principle for each processor.
```

```
int[0,MN] ltog[M][N]={{1, 0}, {2, 3}};//global taskids from locals

broadcast chan reschedule; //broadcast channel for reschelduling after a task has
    finished
chan trigger[MN];

chan activateReschedule;
chan cact[M][3]; //channel array for internal processor actions
chan sact[M][4]; //channel array for scheduling actions
bool preempted; //set when any task is preempted

int[0,N] tid[M]; //transfer of local taskid from task to controller
int[0,N] curtid[M]; //variable used to hold the id of the task currently chosen
int[0,MN] gtid[M]; //transfer of global taskid from task to controller
int[0,MN] ftid[M]; //taskid of task which has finished, ftid=0 means no finished
    task
int[0,MN] ltid[M]; //variable used to hold the id of the task currently running
bool Ready[M][N]; //array of tasks which have issued ready signals
bool Synchronized[MN]; //array of tasks which are not awaiting dependencies to be
    resolved
bool Allocated[MN]; //array of tasks which are not awaiting resources
bool taskReadyForTriggering[MN]; //array holding tasks ready for triggering
bool preemptedOrIdle[MN]; //Array used in the cost model to test if a task is
    preempted or idle
//Array containing the needed resources (as booleans)
bool NeededResources[M][N][R] = {{{0},{0}},{{0},{0}}};

int UsedResources[M][R]; //array indicating whether a resource is taken

bool WaitDep[MN]; //array for tasks which are awaiting dependencies to be resolved

bool running[M]; //indicating wether a task is currently running on the processor

bool missedDeadline = false; //Set if a hard-deadline task misses a deadline.
bool endState[MN]; //Set when a task reaches its end state.

int totalCost[M][NOC] = {{0,0},{0,0}};

//[task1..taskMN][mem, power][static, idle, ready, run, preempt]
int costs[MN][NOC][5] =
    {{{0,0,0,0,0},{0,0,0,0,0}},{{0,0,0,0,0},{0,0,0,0,0}},{{0,0,0,0,0},{0,0,0,0,0}}};

//Writer/Reader
int costDepend[MN][MN] = {{0,0,0},{0,0,0},{0,0,0}};


int sharedCost[MN][MN] = {{0,0,0},{0,0,0},{0,0,0}};

//Criteria usable in scheduling
int staticCriteria[M][N]; //criterion used for static scheduling and contains the
    original parameters for dynamic scheduling.
int dynamicCriteria[M][N]; //criterion used for dynamic scheduling

//array for original dependencies, 1 for dependency, 0 for no dependency

bool origdep[MN][MN]=
{{0,0,0},{1,0,0},{0,0,0}};

//dynamically updated array for current dependencies
bool depend[MN][MN]={{0,0,0},{1,0,0},{0,0,0}};

//Locking mechanisms
bool nowrun[M]; //ensuring completion of 'runs' before reacting on ready
bool pending[MN]; //ensuring global wait for finish & ready before next 'run'
bool h_edf[MN]; //ensuring synchronization on extra state in ready for edf
bool h_t[M]; //ensuring completion of all 'runs' before next 'run'
bool h_fin[M]; //ensuring reaction on finish before ready
bool l_in[M]; //ensuring completion of local scheduling before glocal reschedule
bool h_r; //ensuring reschedule before next 'run'
bool controllerLock[M]; //External controller lock
bool processing[M]; //Internal controller lock

//function checking for dependencies for task t
bool taskHasDependency(int t) {
  for (ini : int[0,MN-1]) {
    if (depend[t][ini]) {
      return true;
    }
  }
  return false;
}

//function updating dependencies when task t has finished
void opdDep(int t) {
  for (ini : int[0,MN-1]) {
    depend[ini][t]=false;
```

```
  }
}

//function setting the original dependency values for task t
void setOrigDep(int t) {
  for (ini : int[0,MN-1]) {
    depend[t][ini]=origdep[t][ini];
  }
}

//function checking for existance of boolean value in array of size M
bool locked(bool la[M]) {
  for (ini : int[0,M-1]) {
    if (la[ini]) {
      return true;
    }
  }
  return false;
}

//function checking for existance of boolean value in array of size N
bool pend(bool pen[N]) {
  bool b = false;
  for (ini : int[0,N-1]) {
    if (pen[ini] == true) {
      return true;
    }
  }
  return false;
}

//Function to verify if all tasks has reached their end states
bool allFinish() {
  for (ini : int[0, MN-1]) {
    if (!endState[ini]) {
      return false;
    }
  }
  return true;
}

//function checking for existance of boolean value in array of size MN
bool pendMN(bool pen[MN]) {
  bool b = false;
  for (ini : int[0,MN-1]) {
    if (pen[ini] == true) {
      return true;
    }
  }
  return false;
}

//Function used to check the total cost in any state
int totalCostUsed(int costnr) {
  int total = 0;
  for (ini : int[0,M-1]) {
    total += totalCost[ini][costnr];
  }
  return total;
}

//Function used to see if any ready or finish signal still needs to be recieved
bool notSignalsPending() {
  return (!locked(h_t)
          && !locked(h_fin)
          && !locked(processing)
          && !pendMN(pending)
          && !pendMN(h_edf));
}

//Function that adds cost to the cost used in beginning of a timeunit.
void updateCost(int gtasknr, int schnr, int State) {
  if (State == READY && preemptedOrIdle[gtasknr-1]) {
    State = PREEMPTED;
  }
  for (i :int[0,NOC-1]) {
    if (i==Memory && State == RUNNING) {
      int sharedCostTemp = 0;
      for (j: int[0, MN-1]) {
        sharedCostTemp += sharedCost[gtasknr-1][j];
        sharedCost[gtasknr-1][j] = costDepend[gtasknr-1][j];
      }
      totalCost[schnr-1][i] += costs[gtasknr-1][i][State]-sharedCostTemp;
    }
    else {
      totalCost[schnr-1][i] += costs[gtasknr-1][i][State];
    }
```

```
    }
}

//Function that resets the cost in the end of a timeunit
void resetCost(int gtasknr, int schnr, int State) {
    if (State == READY && preemptedOrIdle[gtasknr-1]) {
        State = PREEMPTED;
    }
    if (!(State == IDLE && !preemptedOrIdle[gtasknr-1])) {
        for (i :int[0,NOC-1]) {
            //Resets the memory_use after first timeunit.
            if (i==Memory && State == RUNNING) {
                int sharedCostTemp = 0;
                for (j: int[0, MN-1]) {
                    if (sharedCost[j][gtasknr-1] > 0) {
                        for (k : int[0, M-1]) {
                            for (l : int[0,N-1]) {
                                if (ltog[k][l] == (j+1)) {
                                    totalCost[k][i] -= sharedCost[j][gtasknr-1];
                                    sharedCost[j][gtasknr-1] = 0;
                                }
                            }
                        }
                    }
                    if (sharedCost[gtasknr-1][j] == costDepend[gtasknr-1][j])
                        sharedCostTemp -= sharedCost[gtasknr-1][j];
                    else
                        sharedCostTemp -= costDepend[gtasknr-1][j];
                }
                totalCost[schnr-1][i] -= costs[gtasknr-1][i][State]+sharedCostTemp;
            }
            else {
                totalCost[schnr-1][i] -= costs[gtasknr-1][i][State];
            }
        }
    }
}
```

F.2 Task

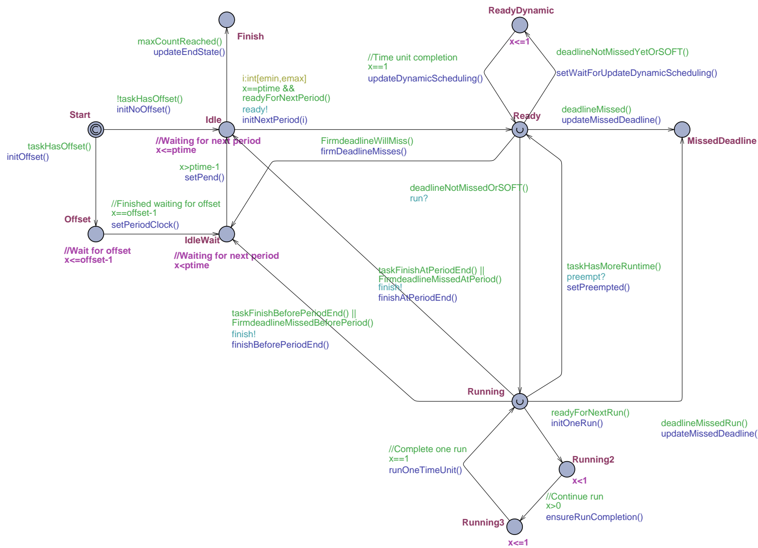

Figure F.1: Task template

F.2.1 Arguments

```
const int schnr, const int gtasknr, const int tasknr, const int emin, const int emax,
const int dead, const int offset, const int ptime, int prior, int[0,N] &tid,
int[0, MN] &gtid, chan &ready, chan &run, chan &preempt, chan &finish,
const int[0,2] sd, int maxCount
```

F.2.2 Local declarations

```
clock x;
int counter = 0;

int cp;
int cr;

int i;

bool taskHasOffset() {
  return (offset >0);
}

bool maxCountReached() {
  return (counter>=maxCount && maxCount > 0);
}

void updateEndState() {
  endState[gtasknr-1]=true;
  pending[gtasknr-1] = false;
}

void firmDeadlineMisses() {
  dynamicCriteria[schnr-1][tasknr-1]=0;
  Ready[schnr-1][tasknr-1]=false;
  Synchronized[gtasknr-1]=false;
  Allocated[gtasknr-1]=false;
  x = cp;
}

void finishBeforePeriodEnd() {
  tid=tasknr;
  gtid = gtasknr;
  dynamicCriteria[schnr-1][tasknr-1]=0;
  h_fin[schnr-1]=false;
  x = cp;
  preemptedOrIdle[gtasknr-1] = true;
  updateCost(gtasknr, schnr, IDLE);
}

void finishAtPeriodEnd() {
  if (!(sd == FIRM && cp==ptime && cr>0)) {
    tid=tasknr;
    gtid=gtasknr;
  }
  dynamicCriteria[schnr-1][tasknr-1]=0;
  h_fin[schnr-1]=false;
  if (counter < maxCount || maxCount == 0) {
    pending[gtasknr-1]=true;
  }
  if (cp > ptime && sd == SOFT) {
    cp = ptime;
  }
  x=cp;
}

void runOneTimeUnit() {
  h_t[schnr-1]=false;
  l_in[schnr-1]=false;

  cr--;
  cp++;
  dynamicCriteria[schnr-1][tasknr-1]--;
  nowrun[schnr-1]=false;

  if (cr==0 || (sd == FIRM && cp==dead)) {
    h_fin[schnr-1]=true;
  }
  resetCost(gtasknr, schnr, RUNNING);
```

```
}

bool taskHasMoreRuntime() {
  return (cr>0);
}

bool deadlineMissed() {
  return (cp==dead && sd == HARD && tid != tasknr);
}

bool deadlineMissedRun() {
  return (cp>dead && sd == HARD);
}

bool deadlineNotMissedYetOrSOFT() {
  return (((cp<dead && sd == HARD)
           || (sd == SOFT)
           || (cp<dead-1 && sd==FIRM))
          && !locked(controllerLock)
          && !locked(h_fin)
          && !locked(h_t)
          && !pendMN(pending));
}

bool deadlineNotMissedOrSOFT() {
  return (((cp<=dead && sd == HARD)
           || (sd == SOFT)
           || (cp<=dead-1 && sd==FIRM))
           && tid == tasknr);
}

bool FirmdeadlineWillMiss() {
  return (sd==FIRM
          && cp == (dead-1)
          && tid != tasknr
          && !locked(controllerLock)
          && !locked(h_fin)
          && !locked(h_t));
}

bool FirmdeadlineMissedBeforePeriod() {
  return (sd==FIRM
          && cp == dead
          && ptime > dead
          && cr > 0);
}

bool FirmdeadlineMissedAtPeriod() {
  return (sd==FIRM
          && cp == ptime
          && cr > 0);
}

void updateMissedDeadline() {
  missedDeadline=true;
}

void setPeriodClock() {
  x=ptime-1;
}

void initOneRun() {
  x=0;
  l_in[schnr-1]=true;
  nowrun[schnr-1]=true;
  preemptedOrIdle[gtasknr-1] = false;
  updateCost(gtasknr, schnr, RUNNING);
}

void setPreempted() {
  preempted=true;
  preemptedOrIdle[gtasknr-1] = true;
  x=0;
}

void initOffset() {
  x=0;
  updateCost(gtasknr, schnr, STATIC);
}

void ensureRunCompletion() {
  h_t[schnr-1]=true;
}

bool readyForNextPeriod() {
  return !(nowrun[schnr-1]
           || locked(h_t)
```

```
            || locked(h_fin)
            || pendMN(h_edf)
            || maxCountReached());
}

void setStaticScheduling() {
  if (processorScheduling[schnr-1] == FP) {
    staticCriteria[schnr-1][tasknr-1]=prior;
  }
  if (processorScheduling[schnr-1] == RM) {
    staticCriteria[schnr-1][tasknr-1]=ptime;
  }
  if (processorScheduling[schnr-1] == DM) {
    staticCriteria[schnr-1][tasknr-1]=dead;
  }
}

void initNextPeriod(int i) {
  cp=0;
  tid=tasknr;
  pending[gtasknr-1]=false;
  dynamicCriteria[schnr-1][tasknr-1]=dead;
  setOrigDep(gtasknr-1);
  cr=i;
  if (!maxCount==0) {
    counter++;
  }
  setStaticScheduling();


  x=0;
}

void setWaitForUpdateDynamicScheduling() {
  h_edf[gtasknr-1]=true;
  updateCost(gtasknr, schnr, READY);
}

void updateDynamicScheduling() {
  dynamicCriteria[schnr-1][tasknr-1]--;
  x=0;
  cp++;
  h_edf[gtasknr-1]=false;
  resetCost(gtasknr, schnr, READY);
}


void initNoOffset() {
  cp=ptime;
  pending[gtasknr-1]=true;
  updateCost(gtasknr, schnr, STATIC);

  x=ptime;
}

void setPend() {
  if (counter < maxCount || maxCount == 0) {
    pending[gtasknr-1]=true;
  }
  resetCost(gtasknr, schnr, IDLE);
  preemptedOrIdle[gtasknr-1] = false;
}

bool taskFinishBeforePeriodEnd() {
  return (cr==0
          && !locked(h_t)
          && !pendMN(h_edf)
          && cp<ptime
          && (cp<=dead
               || (cp>dead && sd==SOFT)));
}

bool taskFinishAtPeriodEnd() {
  return (cr==0
          && !locked(h_t)
          && !pendMN(h_edf)
          && ((cp==ptime &&cp==dead)
                    || (cp > ptime && sd == SOFT)));
}

bool readyForNextRun() {
  return (cr>0
          && !pendMN(pending)
          && !locked(h_t)
          && !locked(h_fin)
          && !locked(controllerLock)
          && !h_r
```

```
            && ((sd == SOFT)
                || (cp < dead && sd == FIRM)
                || (cp <=dead && sd == HARD)));
}
```

F.3 Control

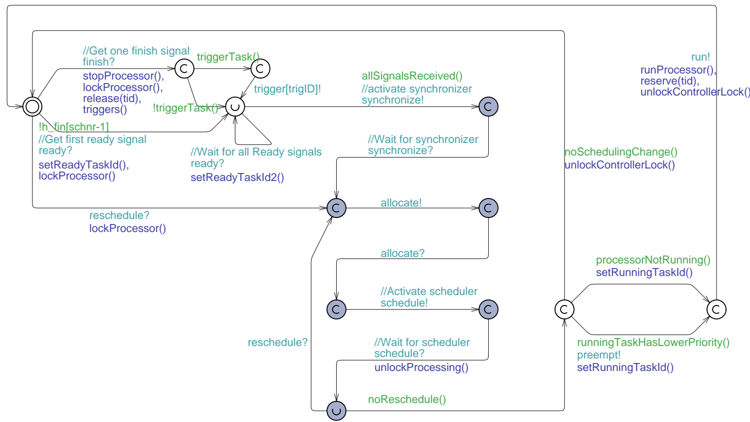

Figure F.2: Control template

F.3.1 Arguments

```
const int schnr, int[0,MN] &ftid, int[0,N] &tid, int[0,MN] &gtid, chan &synchronize,
chan &schedule, chan &allocate, chan &ready, chan &finish, chan &run, chan &preempt
```

F.3.2 Local declarations

```
int[-1,MN] trigID = -1;
int lt = 0;

void triggers() {
  trigID = -1;
  for (it:int[0,MN-1]) {
    lt = ltog[schnr-1][tid-1];
    if (lt != 0) {
      if (origdep[it][lt-1] && taskReadyForTriggering[it]) {
        trigID = it;
      }
    }
  }
}

bool triggerTask() {
  return (trigID >(-1));
}
```

```
bool allSignalsReceived() {
  return (!pendMN(pending) &&
          !locked(h_fin) &&
          !pendMN(h_edf));
}

void stopProcessor() {
  ftid=gtid;
  running[schnr-1]=false;
}

void lockProcessor() {
  controllerLock[schnr-1]=true;
  processing[schnr-1] = true;
}

void unlockProcessing() {
  processing[schnr-1]=false;
}

void unlockControllerLock() {
  controllerLock[schnr-1]=false;
}

void setReadyTaskId() {
  Ready[schnr-1][tid-1]=true;
  ftid=0;
}

bool noReschedule() {
  return (!h_r && !locked(processing));
}

void setReadyTaskId2() {
  Ready[schnr-1][tid-1]=true;
}

bool processorNotRunning() {
  return (!running[schnr-1] && curtid[schnr-1] != 0);
}

void runProcessor() {
  running[schnr-1] = true;
}

void setRunningTaskId() {
  tid=curtid[schnr-1];
  ltid[schnr-1] = curtid[schnr-1];
}

bool noSchedulingChange() {
  return ((ltid[schnr-1]==curtid[schnr-1] && running[schnr-1]) ||
       curtid[schnr-1]==0);
}

bool runningTaskHasLowerPriority() {
  return (ltid[schnr-1]!=curtid[schnr-1] && running[schnr-1]);
}

void reserve(int task){
  for(it:int[0,R-1]){
    if (NeededResources[schnr-1][task-1][it])
      UsedResources[schnr-1][it]=task;
  }
}

void release(int task){
  for(it:int[0,R-1]){
    if (NeededResources[schnr-1][task-1][it])
      UsedResources[schnr-1][it]=false;
  }
}
```

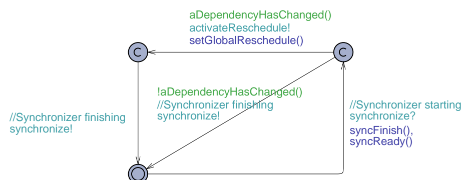

Figure F.3: Synchronizer template

F.4 Synchronizer

F.4.1 Arguments

`const int schnr, int[0,MN] &ftid, chan &synchronize`

F.4.2 Local declarations

```
int[0,MN] i; //iteration variable
int[0,MN] li; //variable used to hold global id for tasks
bool depCh; //flag used if a dependency has been changed


bool aDependencyHasChanged() {
  return (depCh);
}

bool aTaskHasFinished() {
  return (ftid!=0);
}

void setGlobalReschedule() {
  depCh=false;
  h_r = true;
}


void syncFinish() {
  if (ftid > 0) {
    opdDep(ftid-1);
    Synchronized[ftid-1]=false;
    for (i : int[0,MN-1]) {
      if (WaitDep[i] && !taskHasDependency(i)) {
        Synchronized[i]=true;
        WaitDep[i]=false;
        depCh=true;
      }
    }
  }
}

void syncReady() {
  for (i : int[0, N-1]) {
    if (Ready[schnr-1][i]) {
      li=ltog[schnr-1][i];
      Ready[schnr-1][i]=false;
      if (taskHasDependency(li-1)) {
        WaitDep[li-1]=true;
      }
      else {
        Synchronized[li-1]=true;
      }
    }
```

```
  }
}
```

F.5 Allocator

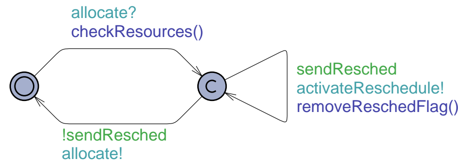

Figure F.4: Allocator template

F.5.1 Arguments

```
const int schnr, int allocationProtocol, chan &allocate
```

F.5.2 Local declarations

```
int gti;
bool resourceConflict,sendResched;

void removeReschedFlag() {
  sendResched=false;
}

void checkResources(){
  for(it:int[0,N-1]){
    gti=ltog[schnr-1][it];
    if (gti){
      if(Synchronized[gti-1]){
        for(r:int[0,R-1]){
          if (UsedResources[schnr-1][r] && allocationProtocol==NPCS) {
            resourceConflict = true;
          }
          else if(NeededResources[schnr-1][it][r] && UsedResources[schnr-1][r]){
            resourceConflict=true;
            if(allocationProtocol==PRI_INH &&
                 (staticCriteria[schnr-1][(UsedResources[schnr-1][r])-1] >
                 staticCriteria[schnr-1][it])){
              staticCriteria[schnr-1][(UsedResources[schnr-1][r])-1] =
                   staticCriteria[schnr-1][it];
              h_r=true;
              sendResched=true;
            }
            if(allocationProtocol==PRI_INH &&
                 (dynamicCriteria[schnr-1][(UsedResources[schnr-1][r])-1] >
                 dynamicCriteria[schnr-1][it])){
              dynamicCriteria[schnr-1][(UsedResources[schnr-1][r])-1] =
                   dynamicCriteria[schnr-1][it];
              h_r=true;
              sendResched=true;
            }
          }
        }
        if(!resourceConflict)
          Allocated[gti-1]=true;
        resourceConflict=false;
```

```
      }
      else{
        Allocated[gti-1]=false;
      }
    }
  }
}
```

F.6 Scheduler

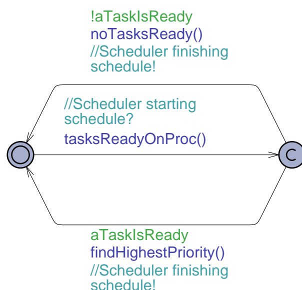

Figure F.5: Scheduler template

F.6.1 Arguments

`const int schnr, int[0,MN] &ftid, chan &schedule`

F.6.2 Local declarations

```
int[0,N] it; //iteration variable
int lcri; //variable used to hold the criterion of the task currently chosen
int[0,MN] li; //variable used to hold global id for tasks
int transID=0;

bool aTaskIsReady;

void noTasksReady() {
  curtid[schnr-1]=0;
}

void findHighestPriority() {
  for (it : int[0,N-1]) {
    li = ltog[schnr-1][it];
    if (li!=0) {
      if (Allocated[li-1]) {
        if (processorScheduling[schnr-1] != EDF && (staticCriteria[schnr-1][it] <
            lcri)) {
          curtid[schnr-1]=it+1;
          lcri=staticCriteria[schnr-1][it];
        }
        else if (processorScheduling[schnr-1] == EDF &&
            (dynamicCriteria[schnr-1][it] < lcri)) {
          curtid[schnr-1]=it+1;
          lcri=dynamicCriteria[schnr-1][it];
        }
```

```
            }
        }
    }
}

void tasksReadyOnProc() {
    aTaskIsReady = false;
    for (it : int[0,N-1]) {
        li = ltog[schnr-1][it];
        if (li != 0) {
            if (Allocated[li-1]) {
                aTaskIsReady = true;
                if (processorScheduling[schnr-1] != EDF) {
                    lcri = staticCriteria[schnr-1][it];
                }
                else {
                    lcri = dynamicCriteria[schnr-1][it];
                }
                curtid[schnr-1] = it+1;
                return;
            }
        }
    }
}
```

F.7 Rescheduler

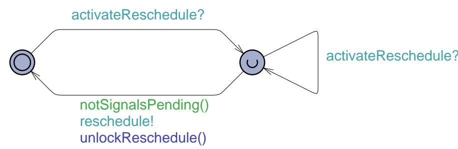

Figure F.6: Rescheduler template

F.7.1 Arguments

None

F.7.2 Local declarations

```
int[0,N] it; //iteration variable
int lcri; //variable used to hold the criterion of the task currently chosen
int[0,MN] li; //variable used to hold global id for tasks
int transID=0;

bool aTaskIsReady;

void noTasksReady() {
    curtid[schnr-1]=0;
}

void findHighestPriority() {
    for (it : int[0,N-1]) {
        li = ltog[schnr-1][it];
        if (li!=0) {
            if (Allocated[li-1]) {
                if (processorScheduling[schnr-1] != EDF && (staticCriteria[schnr-1][it] <
                        lcri)) {
```

```
            curtid[schnr-1]=it+1;
            lcri=staticCriteria[schnr-1][it];
          }
          else if (processorScheduling[schnr-1] == EDF &&
               (dynamicCriteria[schnr-1][it] < lcri)) {
            curtid[schnr-1]=it+1;
            lcri=dynamicCriteria[schnr-1][it];
          }
        }
      }
    }
}

void tasksReadyOnProc() {
  aTaskIsReady = false;
  for (it : int[0,N-1]) {
    li = ltog[schnr-1][it];
    if (li != 0) {
      if (Allocated[li-1]) {
        aTaskIsReady = true;
          if (processorScheduling[schnr-1] != EDF) {
            lcri = staticCriteria[schnr-1][it];
          }
          else {
            lcri = dynamicCriteria[schnr-1][it];
          }
          curtid[schnr-1] = it+1;
          return;
      }
    }
  }
}
```

F.8 Triggered

F.8.1 Arguments

`const int schnr, const int gtasknr, const int tasknr, const int emin, const int emax, const int dead, int prior, int[0,N] &tid, int[0,MN] >id,chan &trigger, chan &ready, chan &run, chan &preempt, chan &finish, const int[0,2] sd`

F.8.2 Local declarations

```
clock x;
int counter = 0;

int cp;
int cr;

int i;

void setReadyForTriggering() {
  taskReadyForTriggering[gtasknr-1] = true;
  endState[gtasknr-1] = true;
  updateCost(gtasknr, schnr, STATIC);
  resetCost(gtasknr, schnr, IDLE);
  preemptedOrIdle[gtasknr-1] = false;
}

void finished() {
  if (!(sd==FIRM && cp==dead && cr > 0)) {
    tid=tasknr;
    gtid=gtasknr;
  }
  endState[gtasknr-1]=true;
  h_fin[schnr-1] = false;
  updateCost(gtasknr, schnr, IDLE);
  preemptedOrIdle[gtasknr-1] = true;
}
```

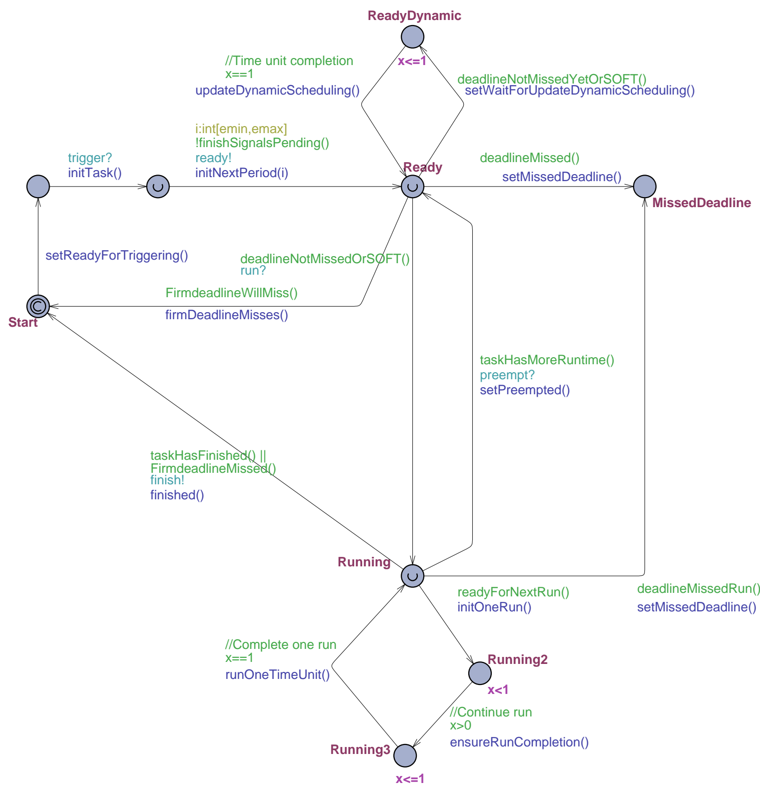

Figure F.7: Triggered template

```
void assignEnoughRuntime() {
  cp=dead-cr;
  x=0;
}

void runOneTimeUnit() {
  h_t[schnr-1]=false;
  l_in[schnr-1]=false;

  cr--;
  cp++;
  dynamicCriteria[schnr-1][tasknr-1]--;
  nowrun[schnr-1]=false;
  resetCost(gtasknr, schnr, RUNNING);
  if (cr==0 || (sd == FIRM && cp==dead)) {
    h_fin[schnr-1]=true;
  }
}

bool taskHasMoreRuntime() {
  return (cr>0);
}

bool deadlineMissed() {
  return (cp==dead && sd == HARD && tid != tasknr);
}

bool deadlineMissedRun() {
```

```
  return (cp==dead && sd == HARD && cr > 0);
}

bool deadlineNotMissedYetOrSOFT() {
  return (((cp<dead && sd == HARD)
          || (sd == SOFT)
          || (cp<dead-1 && sd==FIRM))
          && !locked(controllerLock)
          && !locked(h_fin)
          && !locked(h_t));
}

bool deadlineNotMissedOrSOFT() {
  return (((cp<=dead && sd == HARD)
          || (sd == SOFT)
          || (cp<=dead-1 && sd==FIRM))
           && tid == tasknr);
}

bool FirmdeadlineWillMiss() {
  return (sd==FIRM
          && cp == (dead-1)
          && tid != tasknr);
}

bool FirmdeadlineMissed() {
  return (sd==FIRM &&
          cp==dead &&
          cr > 0);
}

void setMissedDeadline() {
  missedDeadline=true;
}

void initOneRun() {
  x=0;
  l_in[schnr-1]=true;
  nowrun[schnr-1]=true;
  preemptedOrIdle[gtasknr-1] = false;
  updateCost(gtasknr, schnr, RUNNING);
}

void setPreempted() {
  preempted = true;
  preemptedOrIdle[gtasknr-1] = true;
  x=0;
}

void ensureRunCompletion() {
  h_t[schnr-1]=true;
}

bool finishSignalsPending() {
  return (nowrun[schnr-1]
          || locked(h_fin)
          || pendMN(h_edf));
}

void initNextPeriod(int i) {
  cp=0;
  tid=tasknr;
  gtid = gtasknr;
  pending[gtasknr-1]=false;
  endState[gtasknr-1] = false;
  dynamicCriteria[schnr-1][tasknr-1]=dead;
  setOrigDep(gtasknr-1);
  cr=i;
  x=0;
}

void setWaitForUpdateDynamicScheduling() {
  h_edf[gtasknr-1]=true;
  updateCost(gtasknr, schnr, READY);
}

void updateDynamicScheduling() {
  dynamicCriteria[schnr-1][tasknr-1]--;
  x=0;
  cp++;
  h_edf[gtasknr-1]=false;
  resetCost(gtasknr, schnr, READY);
}


void initTask() {
  pending[gtasknr-1]=true;
```

```
  if (processorScheduling[schnr-1] == FP) {
    staticCriteria[schnr-1][tasknr-1]=prior;
  }
  else {
    staticCriteria[schnr-1][tasknr-1]=dead;
  }
  taskReadyForTriggering[gtasknr-1] = false;
}

bool taskHasFinished() {
  return (cr==0
          && !locked(h_t)
          && !pendMN(h_edf));
}

bool readyForNextRun() {
  return (cr>0
          && !pendMN(pending)
          && !locked(h_t)
          && !locked(h_fin)
          && !locked(controllerLock)
          && !h_r
          && ((sd == SOFT)
               || (cp < dead && sd == FIRM)
               || (cp <=dead && sd == HARD)));
}

void firmDeadlineMisses() {
  dynamicCriteria[schnr-1][tasknr-1]=0;
  Ready[schnr-1][tasknr-1]=false;
  Synchronized[gtasknr-1]=false;
  Allocated[gtasknr-1]=false;
  updateCost(gtasknr, schnr, IDLE);
  preemptedOrIdle[gtasknr-1] = true;
  x = cp;
}
```

F.9 Environment

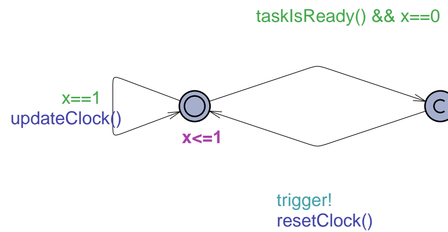

Figure F.8: Environment template

F.9.1 Arguments

```
int task_id, chan & trigger, int interarrival
```

F.9.2 Local declarations

```
int temp;
```

```
clock x;
int cl;

bool taskIsReady() {
  return (!locked(h_t)
          && !locked(l_in)
          && taskReadyForTriggering[task_id -1]
          && cl >=interarrival);
}

void updateClock() {
  x=0;
  if (cl < 32000) {
    cl++;
  }
}

void resetClock() {
  x=0;
  cl=0;
}
```

Appendix G

Contents of the CD-rom

The attached CD to this thesis contains the following:

- Source code for the MoVES tool is found in the folder `MoVES`.
- Source code for the examples described in Chapter 5, is found in the directory `Testfiles`.
- The API for the MoVES tool is located in the directory `API`. This gives an description of the different functions and variables in each class of the tool.
- The special Verifyta provided by Jacob Illum, which is described in Section 3.4, is found in the directory `Special_Verifyta`. The Verifyta works under Linux only.
- Poster, brochure and abstract for DATE'07 is located in the folder `Date`.
- Slideshow presented at the MoDES workshop is found in `MoDES`
- An electronic version of this thesis in PDF format can be found in the directory `Thesis`.
  - Impress-files of schedules are located in the subdirectory `Figures`.

Bibliography

[1] Gregory R. Andrews. *Foundations of multithreaded, parallel, and distributed programming.* Addison-Wesley, Reading, Mass. [u.a.], 2000.

[2] Sanjoy K. Baruah, Rodney R. Howell, and Louis E. Rosier. Feasibility problems for recurring tasks on one processor. *Theoretical Computer Science*, 118(1):3–20, 1993.

[3] 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.

[4] A. Brekling. Modelling and verification of mpsoc. Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, 2006. Supervised by Associate Professor Michael R. Hansen and Professor Jan Madsen, IMM.

[5] A. Brekling, M. R. Hansen, and J. Madsen. A timed-automata model of multiprocessor system-on-chips. 2007.

[6] G. Buttazzo, G. Lipari, L. Abeni, and M. Caccamo. *Soft Real-Time Systems: Predictability vs. Efficiency (Series in Computer Science).* Plenum Publishing Co., 2005.

[7] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications.

ACM Transactions on Programming Languages and Systems, 8(2):244–263, 1986.

[8] J. Ellebæk, K. S. Knudsen, A. Brekling, M. R. Hansen, and J. Madsen. MOVES - a tool for modeling and verification of embedded systems. In *DATE'07 University Booth*, apr 2007.

[9] 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.

[10] J. Hagman. Mpeg3play-0.9.6. source code.

[11] Jane W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000. LIU j 00:1 1.Ex.

[12] A. 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 2004. ISBN-10: 1-4020-7835-8, ISBN-13: 978-1-4020-7835-4.

[13] 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, 2004.

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

[15] 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, New York, NY, USA, 2000. ACM Press.

[16] J. Regehr. Using hierarchical scheduling to support soft real-time applications on general-purpose operating systems, 2001.

[17] Jan Reineke, Björn Wachter, Stefan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In Frank Mueller, editor, *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006. <http://drops.dagstuhl.de/opus/volltexte/2006/671> [date of citation: 2006-01-01].

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

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

[20] 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.

[21] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systeme, 2000.

[22] Dakai Zhu. Reliability-aware dynamic energy management in dependable embedded real-time systems. *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, pages 397–407, 2006.