Performance Analysis of Distributed Real-Time Embedded Systems

Master thesis

M.M.C.M. de Hoon (535078)

22nd December 2005

Performance Analysis of Distributed Real-Time Embedded System

M.M.C.M. de Hoon

Final report for Master of science project
conducted from December 2004 - December 2005

Department of Electrical Engineering
Information and Communication Systems/Electronic Systems
(ICS/ES)
Technische Universiteit Eindhoven

Professor:
Prof.dr.ir. R.H.J.M. Otten (TU/e)

Supervisors:
Dr. ir. J.P.M. Voeten (TU/e)
M. Sc. O. Florescu (TU/e)

Abstract

The design of a distributed real-time embedded system is a difficult job. The hardware and software is often designed sequentially, leading to overly conservative and expensive systems. A more reliable and optimal system is obtained by introducing performance analysis in the early design phases. This analysis is performed with models designed in an ad-hoc way.
We propose a method which uses models to analyse distributed real-time embedded systems that capture both functional and timing properties, in a early design phase. The models are based on SHE (Software/Hardware Engineering). SHE is a system-level design methodology based on the formal modelling language POOSL (Parallel Object-Oriented Specification Language), and on the fast execution engine Rotalumis. The modelling method is based on the Y-chart scheme and involves specification of the environment, the application, the architecture and the mapping between them. This thesis presents modelling patterns for common input/output devices, real-time tasks and platform resources. With these patterns a model of a distributed real-time embedded can be build conveniently. The patterns are validated by means of a realistic case study of an in-car navigation system. The outcome of the performance analysis is compared with the outcome of a Modular Performance Analysis (MPA), a method based on worst-case execution analysis. The comparison shows that the proposed method produce performance numbers that approximate the worst case execution as opposed to MPA which is sometimes overly conservative. The proposed method effectively captures the behaviour of both soft and firm real-time embedded systems.

Acknowledgements

I hereby thank the people from the TU/e ICS/ES department for giving me this opportunity. I would also like to thank Jeroen Voeten, my supervisor, for giving me the opportunity to develop my own ideas. My special thanks goes to Oana Florescu for coaching my during this thesis. From the early stages until the final version she was always able to help me structure my thesis and gave me helpful feedback to improve my work. I also want to thank Marcel Verhoef, a member of the Embedded System Institute, for giving me the In-Car navigation system case study.

Finally, I wish to express my thanks to my family and friends, who have supported me during my time on the TU/e and my graduation. Especially my parents for giving me the chance to continue my study at the University and their support in reaching this goal.

Eindhoven, December 2005
Menno de Hoon

Contents

List of Figures

Chapter 1

Introduction

The real-time embedded systems industry today must realise its product ideas even quicker than in the past. To be competitive, these new real-time embedded systems must support more functionality, make use of latest technical innovations and, of course, must be low cost. Real-time embedded systems which support much functionality are complex and hard to design. The industry often uses methods to specify hardware and software separately, often leading to overly conservative systems. Overly conservative systems largely contribute to the product cost. One of the reasons is the lack of a proper modelling methodology to give insight in the behaviour of the system, which would help in finding the optimal hardware and software combination. A modelling methodology enables modelling of complex real-time embedded systems and provides insight in the behaviour of the system in the early design phases. Such modelling methodologies must take both the software and the hardware part of embedded system into account. This eventually must result in a reliable and optimal embedded system, designed in less time.

1.1 Problem definition

Existing design methods, for instance object-oriented design methods, focus on reusing and maintaining large systems. These design methods have proven their benefit especially for traditional software development. However these methods are not adequate for designing real-time embedded systems. A design method for real-time embedded system should provide a modelling technique that can appropriately capture functional and timing properties. A design method called Software/Hardware Engineering (SHE) is presented in [1]. SHE is a system-level design methodology based on the formal modelling language POOSL (Parallel Object-Oriented Specification Language), and on the fast execution engine Rotalumis. The POOSL models can be specified with the graphical tool SHESim. The methodology allows specification and analysis of real-time discrete-event control systems, such as a high-speed packet-switch, a network processor, a printer controller and a wafer-stepper controller. The specification of these system is done ad-hoc. A suitable way to model and analyse these kinds of systems is necessary. Therefore a modelling method should provide an approach to model a system in an adequate way. To speed up the design process a method should be supported by a library. This library must contain components that have common characteristics of a real-time embedded system. To simplify the design space explorations these patterns must be modular (plug-and-play).

1.2 Objectives

The objectives in this thesis to cope with problem definition are given in the following enumeration:

1. **Develop a modelling method which is suitable for performance analysis and design space exploration of distributed real-time embedded systems.** A modelling method should be defined which helps to design a model of a real-time embedded system. The model needs to capture both the functional and timing behaviour and should be suitable for performance analysis. This method should be supported with a library which consists of a basic set of components that capture common characteristics of a real-time embedded system. The use of predefined components must speed up the modelling process. A modular design approach should simplify the design space exploration.

2. **Show applicability of the modelling method.** The applicability of the method should be demonstrated by an industrial case study. A performance analysis should be used to validate the components of the modelling method.

1.3 Main contributions

During the project we developed a method for modelling distributed real-time embedded systems. This modelling method describes how to capture both functional and timing behaviour. The method is based on the Y-chart. The Y-chart scheme structure a system for design space exploration. For each part of the Y-chart scheme we present components which models common characteristics of real-time embedded systems. The following list presents parts of the Y-chart scheme.

- **Environment:** In the environment section components characterise common input and output devices of a real-time embedded system by generating event patterns.

- **Application:** The application section defines the functional behaviour of a real-time embedded system.

- **Architecture:** Architecture components characterises processor and communication resources. The processor components models the computation with rate monotonic, earliest deadline first or time sliced scheduling. The communication components model data exchange with a first come first serve discipline.

- **Mapping:** In the mapping section we have specifies how an application is mapped on an architecture in a modular way.

- **Performance analysis:** By analysing the combined model the performance properties (throughput, occupation, delay, etc.) can be deduced.

All these components are specified in a modular way (plug-and-play) which simplifies the exploration of the design space. The method was validated by means of a realistic industrial case study. The outcome of the performance analysis is compared with the outcome of a Modular Performance Analysis (MPA), a method based on worst-case execution analysis. The comparison showed that POOSL performance numbers approximate the worst case execution and MPA is sometimes overly conservative. As the POOSL analysis technique relies on simulation; the discovery of the worst case execution can not be claimed. Moreover,

the accuracy of the performance results depends on the simulation length. However the advantage is that the realistic behaviour of the system can be captured using distributions.

Chapter 2

Modelling approach

2.1 Introduction

Real-time embedded systems are difficult to design. They consist of both hardware and software components. The technological advance and the demand for more functionality make these systems more complex. The software behaviour in a real-time embedded system depends on the system hardware (architecture). Typically, software and hardware design methodologies are applied in isolation, which, after the combination of their results, result in an over-dimensioned or even non-working system. Some of the classical design methodologies are:

- Structured analysis and design methods (Ward and Mellor [2], Hatley and Pribhai [3])
- Object-oriented and object-based analysis and design methods (UML [4], ROOM [5], etc.)
- Formal description methods (SDL [6], Estelle [7])

For a complete comparison of these methodologies see [1] and [8]. These classical design methodologies do not often adequate help the design process in considerate design alternatives for realising the desired functionality. Early in the design process, the choice for a specific design alternative may have a deep impact on, for example, the performance of the final implementation. To assist the designer in taking well-founded design decisions, system-level design methodologies can be applied. A system-level methodology which supports the construct models that allows the analysis of the system in the early design phase is very helpful.

A suitable modelling methodology which can be used for modelling real-time embedded systems is the Software/Hardware Engineering (SHE) introduced in [1] and briefly described in section 2.2. This modelling methodology has proven its usefulness in modelling several kinds of real-time embedded systems, like [9], [10] and [11]. The designer experienced several disadvantages during the modelling process, such as; a long modelling time, the low degree of reusability and that each model is complex. A reason for this is that these models are modelled without applying a method. The advantages of applying a method for modelling a real-time embedded system are:

- **Reducing the modelling time.** The construction of a real-time embedded system is done by the use of components. The modelling time can be reduced when the designer is able to re-use earlier designed components. To overcome inconsistency these components must comply with the (interface) specifications described in the upcoming chapters.

- **Understandability.** A predefined subdivision of how to model a real-time embedded system model will increase readability.
- **Analysability.** A method is taking care of the possibility to perform a system analyse, for example the performance or occupation.
- **Assist the designer in taking well-founded design decisions.** Applying a method that uses components and predefined system subdivision allows a Design Space Exploration (DSE). A DSE helps the designer to take well-founded design decisions.

In this thesis a modelling method is described to improve the modelling process. The method is extended with several examples of components which can be used for modelling (distributed) real-time embedded systems. These components are specified in the expressive modelling language POOSL, formalised in [1]. A short description of POOSL is given in section 2.3. The tools used for specification and execution of the POOSL components, namely SHEsim and Rotalumis, are briefly discussed in section 2.4. In section 2.5 and 2.6 the guidelines of the modelling method, which is the content of this thesis.

2.2 Software/Hardware Engineering (SHE)

SHE is a system-level design methodology, as defined in [12], that allows analysis of both correctness and performance properties of design alternatives based on models. To construct such models, SHE uses Parallel Object-Oriented Specification Language (POOSL) to formulate and formalise the behaviour of a system. The actual evaluation is based on the application of several techniques for formal verification of correctness properties and performance analysis. A key feature of the SHE methodology is that it is based on formal methods which ensures that the obtained analysis results are unambiguous.

2.3 Parallel Object Oriented Specification Language

In this section, we present a brief overview of the POOSL (Parallel Object Oriented Specification Language) language, which was developed at Eindhoven University of Technology. POOSL is a very expressive modelling language with a small set of powerful primitives whose semantics are defined with mathematical axioms and rules. POOSL can describe concurrency, distribution, communication, timing and functional features of a system in a single executable model. POOSL consists of a process part and a data part. The process part (processes and clusters) is based on a real-time extension of the process algebra CCS [13]. This part is specified in components which performs certain functionality of a system. The data part are passive components that specify the information that is generated, exchanged, interpreted or modified by the system. The data part is based upon the concepts of traditional sequential object-oriented programming languages like Smalltalk and C++. The execution of a POOSL model is based on a two phases execution model [14], as shown in figure 2.1. The state of a model can either change by asynchronously executing atomic (communication or data processing) actions (taking no time) or by letting the time pass (synchronously).

The formal semantics of POOSL enable the application of model checking techniques for formal verification of correctness properties and Markov-chain based performance analysis techniques. Furthermore, it serves as basis for a timing property-preserving approach for real-time software synthesis.

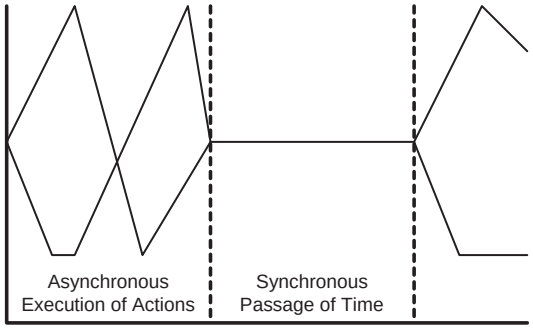


Figure 2.1: Two execution phases of a POOSL model.

2.4 Tools

2.4.1 SHEsim

SHESim is an interactive modelling and simulation tool, which enables the construction of complex concurrent systems in accordance with the SHE methodology. It is used to incrementally specify and modify POOSL data classes, process classes and cluster classes. SHESim allows the (graphical) entry of POOSL models and their interactive simulation. The messages and parameters that are passed between the different processes and clusters are indicated on the appropriate channels. To inspect the history of messages that have been exchanged between different entities, interaction diagrams can be generated automatically during a simulation. For more information see [15].

2.4.2 Rotalumis

Rotalumis is a high-speed execution engine which allows fast simulations of POOSL models. In comparison with the execution speed of the SHEsim tool where the execution takes place in an interpretive way, the execution speed is improved by a factor of 100. Rotalumis compiles the POOSL model into intermediate format that is executed on a virtual machine implemented in C++. For more information see [16]. This academic tool was used for the simulation of all models presented in this thesis. In general, the models are validated in the SHEsim tool and then executed in Rotalumis.

2.5 Modelling Method

The guideline of the modelling method is based on the Y-chart scheme structure. As described in [17], the Y-chart with one extension involves the following:

- **Environment:** Specify the environment behaviour capturing the characteristics of the surroundings, such as input and output devices connected to the real-time embedded system.
- **Application:** An abstraction of the software is defined in the application section. The environment is linked to a set of tasks in the application section. The environment triggers this set of tasks.
- **Architecture:** The modeler describes a particular architecture of the real-time embedded system.

- **Mapping:** In this section the application is mapped on the architecture.
- **Performance analysis:** The mapped architecture and application model are used for performance analysis.
- **Performance numbers:** This analysis yields performance numbers which can propose improvements in the architecture, application and/or mapping. This processes is indicted in figure 2.2 by the light bulbs.

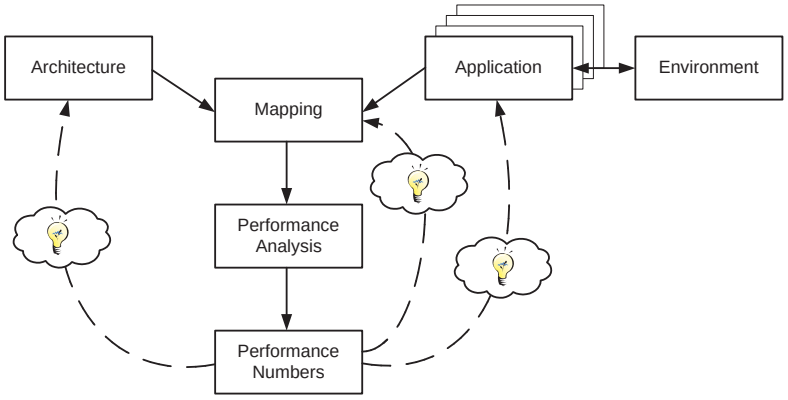


Figure 2.2: The Y-chart design approach.

This procedure can be repeated in an iterative way until a satisfactory architecture, set of application and mapping is found. To be able to use the Y-chart approach, the following modelling steps must be followed:

1. Specify the environment components.
2. Specify the application.
3. Specify the architecture components.
4. Map application components onto the architectural components.
5. Analyse the performance.

The specification of the functional part of the real-time embedded system is described in steps 1 and 2, where the specification of the environment and application models are made. The functional part is "independent" of the architecture specified in step 3. Step 4 is to map the application to the architectural components. After applying the performance analysis, steps 2 till 4 can be reconsidered for optimisation of the system. This modelling process gives the engineer a structured framework to explore the design space of a computation intensive real-time embedded system.

2.6 Report Structure

The structure of this report follows the earlier presented Y-chart scheme, see figure 2.3. The functional model which specifies the environment and application components, is described in chapter 3. The application components require architectural components for execution. The architectural modelling is described in chapter 4. The mapping of the application components onto the architectural components is discussed in chapter 5. Chapter 6 shows the utilisation of the described design approach in a case study. Chapter 7 provides the conclusions and future work related to this thesis.

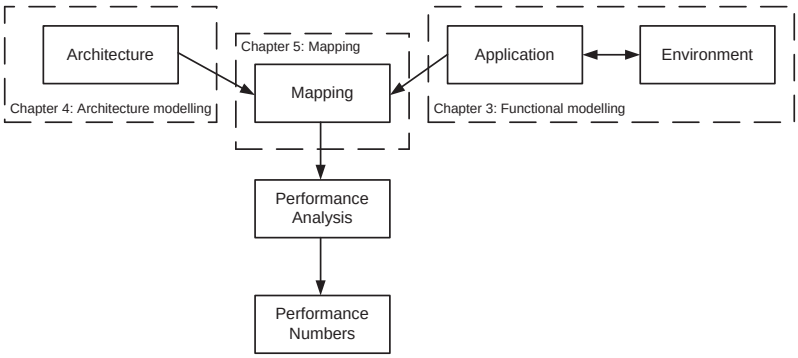


Figure 2.3: Structure of the report based on the Y-chart.

Chapter 3

Modelling of Functional Characteristics

3.1 Introduction

A real-time embedded system performs software tasks that are executed on processors. These software tasks are activated by the working environment of the system. The software tasks and the working environment belong to the functional part of a real-time embedded system model. Figure 3.1 visualises an example of a real-time embedded system in a working environment which helps engineers reason about the total system behaviour. **Note:** These graphical representations are

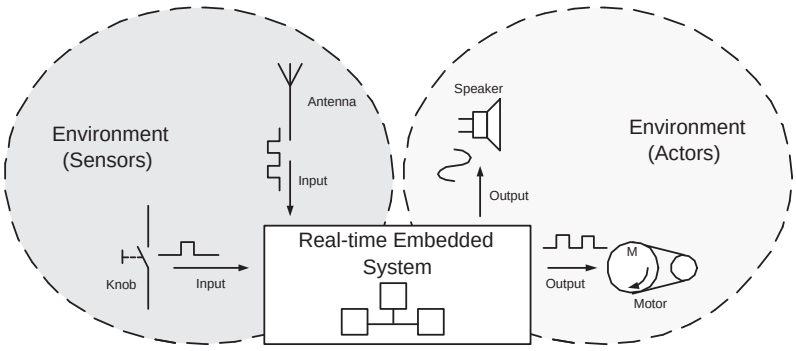


Figure 3.1: Simple view of a real-time embedded system in an environment.

not restricted by drawing rules; their purpose is to clarify the system and working environment.

To model the functional characteristics of a real-time embedded system, the design consists of environment and application components. The environment components model the characteristics of the environment given by input and output devices. Some examples of such devices are sensors, knobs, antennas, motors, speakers and displays. The application components model the software part of a real-time embedded system which are presented as a directed graph of tasks. Event activation patterns generated by the environment components are used for modelling the behaviour of input devices connected to the real-time embedded system.

This chapter is organised as follows:

- Subsection 3.2 describes an approach to model the environment of a real-time embedded system, by use of several kinds of event patterns.
- Subsection 3.3 presents a pattern to model an application, which reflects the software part of the system.

3.2 Environment Modelling

The functional characteristics of the environment are specified in environment components. Environment component models the generation or consumption of event streams. In an event stream each event has a specific time at which it must occur. These event streams are specified as having periodic or sporadic patterns. [18] and [19] define a set of such general event patterns. Important patterns for analysing performance of a real-time embedded systems are events that occur sporadically, periodically (with jitter) or within a burst. Our modelling approach uses a data object to exchange event-related information such as timing variables. This event-related information can be updated during simulation. The specification of this data class is described in section 3.2.1. Sections 3.2.2, 3.2.3 and 3.2.4 describe possible event patterns which commonly occur in real-time embedded system and are suitable for modelling. Finally, section 3.2.5 describes a modelling component which consumes the event stream passed through the system model.

3.2.1 Registering of Timing Properties

The complete model is used to predict the performance of a real-time embedded system. A data class which registers performance properties is used in this modelling method. A new data object is initialised each time when an environment component generates an event. At this moment the release time of the event is registered in this data object. When the environment component triggers the application model (described later on) by an event the start time is registered in the data object. During the execution of the application model the data object is exchanged between tasks. Each task updates the communication or computation variable when it is involved with in, respectively, communication or computation. Finally, the data object will reach a consuming environment component which register the finish time.
POOSL allows the creation of data objects, which are instances of data classes, for modelling passive components (see [20] for more details). Each time when an environment component generates an event, it exchanges this data by sending the data object (making a Deepcopy) along with a message to another component. A characterisation of the timing properties is given in figure 3.2. In this illustration,

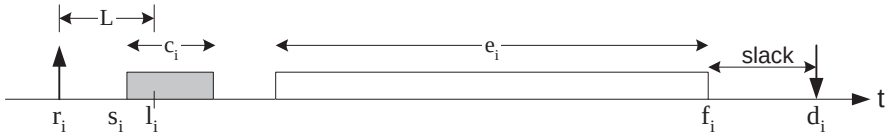


Figure 3.2: Timing properties of events.

`i` stands for the identification number of the event. A description of the illustrated properties are given below:

Release time r_i: is the time at which an event becomes ready for execution;

Start time s_i: is the time at which a task start its execution;

End of release l_i: $l_i = r_i + L$ is the time when the event is not able to trigger a task anymore;

Event Lifetime L: is the amount of time an event is active and able to trigger a task;

Finish time f_i: is the time at which an event finishes its execution;

Communication time c_i: is the time used by the communication link;

Computation time e_i: is the time necessary to the processor for executing the task;

Deadline d_i: is the moment before which a task should be completed to avoid damage to the system;

Slack : X_i: $X_i = d_i - f_i$ is the maximum time an event can be delayed on its release to complete within its deadline;

Note that such a data object is exchanged by several tasks in the application model. Every time the data object is in a new task component, the current values of the communication and computation time are accumulated with the new communication or computation time.
The timing properties are defined in the `EventProperties` data class which modifies and registers the above presented timing properties. Figure 3.3 presents the `EventProperties` data class specified in POOSL. Methods `SetReleaseTime`,

```
<< data class >>
EventProperties : Object
<< instance variables >>
Id : Integer
RelativeDeadline : Real
ReleaseTime : Real
StartTime : Real
FinishTime : Real
ComputationTime : Real
CommunicationTime : Real
<< methods >>
Ini(t : Real) : Object
SetReleaseTime(t : Real) : Object
SetStartTime(t : Real) : Object
SetFinishTime(t : Real) : Object
AddComputationTime(t : Real) : Object
AddCommunicationTime(t : Real) : Object
```

```
1 Ini(t : Real) : Object
2   RelativeDeadline := t;
3   return self.
4 SetReleaseTime(t : Real) : Object
5   ReleaseTime := t;
6   return self.
7 SetStartTime(t : Real) : Object
8   StartTime := t;
9   return self.
10 SetFinishTime(t : Real) : Object
11   FinishTime := t;
12   return self.
13 AddComputationTime
14     (t : Real) : Object
15   ComputionTime :=
16     ComputionTime + t;
17   return self.
18 AddCommunicationTime
19     (t : Real) : Object
20   CommunicationTime :=
21     CommunicationTime + t;
22   return self.
```

Figure 3.3: POOSL specification of the EventProperties data class

`SetStartTime` and `SetFinishTime` specify the release, start and finish time respectively of an event captured in an `EventProperties` data object. When the event data object travels through the application model and reaches a consuming environment component, the timing properties are used for analysis purpose. Methods `AddComputationTime` and `AddCommunicationTime` add repetitively computation and communication time in the data object. The following sections describe several components which generate event patterns that use the `EventProperties` data class.

3.2.2 Modelling Sporadic Event Streams

Sporadic event stream components model the activation of an input device connected to the real-time embedded system. This component is used for modelling devices that are activated irregularly, such as a knob or a remote control. An example of a sporadic event stream is given in figure 3.4. At every r in figure 3.4 an

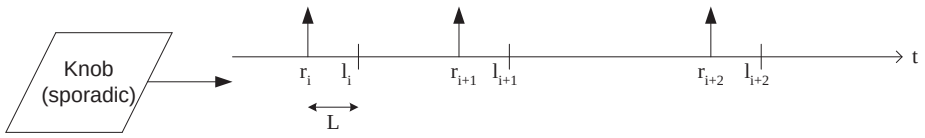


Figure 3.4: Sporadic event stream generated by an event component.

event is released; l denotes the end time of the event lifetime. A task can be triggered by the event between the r and l. The triggering of tasks is only done when the application (the tasks) is capable to serve a new event (the system could be busy). In this way, event misses can be analysed and event releases do not overlap. The specification of a component which produces is given in figure 3.5. In fig-

```
        << process >>
SporadicEventModel
<< instantiation parameters >>
eventLifetime :  real;
   << instance variables >>
t :  RandomGenerator;
        << methods >>
Ini()()
SporadicEventStream()()
   << initial method call >>
Ini()()
        << messages >>
out!event
```

```
1 ini()()
2   t := new(Distribution);
3   SporadicEventStream()().
4 SporadicEventStream()()
5 | E : EventProperties |
6   E := new(EventProperties)
7     SetReleaseTime(currentTime);
8   par
9     abort out!event(E) with
10     delay eventLifetime
11   and
12     delay (t random + eventLiftime);
13     SporadicEventStream()()
14   rap.
```

Figure 3.5: POOSL specification of a sporadic event stream

ure 3.5 `eventLifetime` is an instantiation parameter of the `SporadicEventModel` which specifies the life of an event. The instance `t` is of a distribution type, and used for generating different time between event actuation. In this example a random distribution is used. To guarantee that the occurrences of events do not overlap other events, the specified `eventLifetime` is added to the period of the next released event at line 12.

3.2.3 Modelling Periodic Event Streams

Devices connected to an embedded system that have periodic characteristics, like radio antennas and sensors, are modelled as components that generate events periodically. An example of a periodic event pattern is given in figure 3.6, where r, l and T denote respectively the release time, the end of the event actuation lifetime and the period of the event release. A component which generates a periodic event pattern can be specified in POOSL as described in figure 3.7. As the specification in figure 3.6 describes, `eventLifetime` and `period` are instance parameters that characterise the event activation pattern. To guarantee that every event is activated at the specified time instances, the `PeriodEventStream` is specified as a parallel method. In line 5, an event is offered during the specified

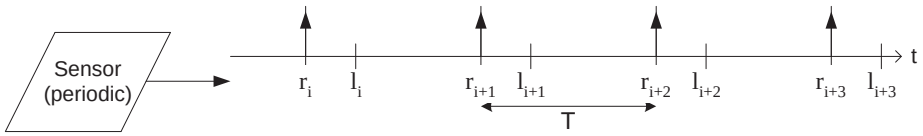


Figure 3.6: Periodic event stream generated by an event component.

```
        << process >>
PeriodicEventModel
<< instantiation parameters >>
eventLifetime :  Real;
period :  Real;
   << instance variables >>
        << methods >>
PeriodicEventStream()()
   << initial method call >>
PeriodicEventStream()()
        << messages >>
out!event
```

```
1 PeriodicEventStream()()
2 | E : EventProperties |
3   E := new(EventProperties) SetReleaseTime(currentTime);
4   par
5     abort out!event(E) with delay eventLifetime
6   and
7     delay period;
8     PeriodicEventStream()()
9   rap.
```

Figure 3.7: POOSL specification of a periodic event stream

`eventLifetime`. When the amount of time specified in `eventLifetime` is elapsed the offering of the event is stopped (an event miss).

3.2.4 Modelling Periodic Event Streams with Jitter

In common distributed real-time embedded systems, input devices produce a fixed number of events in a certain time unit. The exact period between these events is often hard to specify. A component which produces events each period with a jitter is therefore useful. This component is also useful for performance analysis of distributed real-time embedded system where several input devices produce events in different periods. Modelling these input devices with components which generate events periodically will not cover all the states of the system, because combining of periodical events patterns will result in a repetitive occurrence of events. A periodic event pattern with jitter is represented in figure 3.8. In figure

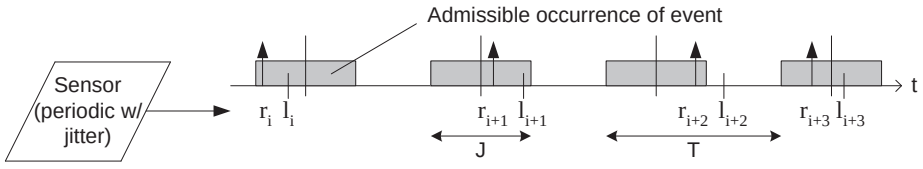


Figure 3.8: Periodic event stream with jitter generated by an event component.

3.10 T, J, r and l denote the period, jitter, release time and end of the release time of an event. The event actuation is between $iT - \frac{1}{2}J$ and $iT + \frac{1}{2}J$. To guarantee no event overlap, the abstract environment model must comply with $T > \frac{1}{2}J + d$. The environment component which generates jittery events uses the specially defined `JitterGenerator` data class, which is specified in figure 3.9. The `next` method in `JitterGenerator` data class returns a value between $-\frac{1}{2}J$ and $+\frac{1}{2}J$, which eases the specification in an environment component. This `JitterGenerator` data class is specified with the use of a distribution (see the instance variables). In this example a random distribution (`RandomGenerator`) is used. An environment component that generates the periodic event stream with jitter is specified in

```
<< data class >>
JitterGenerator : Object
<< instance variables >>
jitter : Real
r : RandomGenerator
<< methods >>
SetJitter(i : Real) : Object
next() : Real
```

```
1 SetJitter(i : Real) : Object
2   jitter := i;
3   return self.
4 next() : Real | n : Real |
5   if jitter > 0 then
6     n := (r random * jitter) - (0.5 * jitter)
7   else
8     n := 0;
9   fi;
10   return n.
```

Figure 3.9: POOSL specification of the jitter generator class

POOSL and shown in figure 3.10. As figure 3.8 shows, the jitter is centralised at

```
<< process >>
PeriodicJitterEventModel
<< instantiation parameters >>
duration : Real;
period : Real;
j : Real;
<< instance variables >>
jitter : JitterGenerator;
<< methods >>
Ini()()
PeriodicJitterEventStream()()
<< initial method call >>
Ini()()
<< messages >>
out!event
```

```
1 ini()()
2   jitter := new(JitterGenerator)
3       SetJitter(j);
4   PeriodicJitterEventStream()().
5 PeriodicJitterEventStream()()
6 | E : EventProperties |
7   par
8     delay period + jitter next;
9     abort out!event(E) with delay duration
10   and
11     delay (period);
12     PeriodicJitterEventStream()()
13   rap.
```

Figure 3.10: POOSL specification of a periodic event stream with jitter

each period. This means that the first event can occur at negative time. An event which occur in negative time can not be modelled. The limitation of this process is that the release time of the first event is equal to or bigger than $T - \frac{1}{2}J$.

3.2.5 Receiving Event Streams

Each event stream that passes through the application model will be received by the environment model. This environment model reflects the actuator devices connected to a real-time embedded system, like motors, displays, speakers, etc. A simple event consuming component is specified in figure 3.11. This tail-recursive

```
<< process >>
EventReceiverModel
<< instantiation parameters >>
<< instance variables >>
<< methods >>
ReceiveEvent()()
<< initial method call >>
ReceiveEvent()()
<< messages >>
in?event
```

```
1 ReceiveEvent()()
2 | E : EventProperties |
3   in?event(E);
4   ReceiveEvent()();
```

Figure 3.11: POOSL specification of a receiving event model

specification of an event consumer component receives event streams from the application model without any restriction. The received event data object `E` is used especially for analysis purposes.

3.3 Application modelling

In the Y-chart scheme, presented in figure 2.2, the environment components trigger the application model. The software behaviour of a real-time embedded system is specified in the application model. The application model consists of nodes which represent tasks of the software. During the modelling phase the actual behaviour of a task does not need to be specified. Finding a good abstraction of a task is hard and typically done by experienced engineers. Characteristics and methods to determine proper abstractions of software task is outside the scope of this thesis. The application model is a directed graph where nodes represent tasks and edges represent activation channels. Note that the edges are not necessary infinite FIFO queues as in the case of Kahn Process Networks, introduced in [21]. A task in this model can also block other tasks. Real-time embedded systems consist not only of processors but also of communication resources. The communication taking place in these resources can be seen as tasks, therefore the application model also exists of components which models the communication in the system. Task components used for modelling the software is described in section 3.3.1. In section 3.3.2 the communication tasks are described.

3.3.1 Software Tasks

In the early design phases, where this modelling approach is used, the software of the embedded system is typically not known in detail. To speedup the modelling process, task components model the main functionality of the software. Representing the application model as a directed graph makes it possible to execute tasks in parallel. When a task is triggered by an event it performs an abstracted software task. These active tasks will trigger new task components to model the complete software. Parameters that involve computation behaviour are defined in the task component, however some general parameters are:

Computation load: each task is specified with a computation load, which can be specified in the amount of cycles or instructions (depending on the type of processor architecture used in the system). When specifying an instruction name it is possible to retrieve the computation load out of a predefined computation table specified in the resource component (described in chapter 4);

Task identifier: a unique identifier in the application graph, which can be a name or a number. The identifier is used to follow the order in which computations are carried out;

Priority / relative deadline [optional]: this number is used for scheduling the computations of tasks;

Note that these parameters depend on the resources of the system. Specifying these parameters at the resource components (see chapter 4) will make the model less suitable for exploration of design alternatives (each task must be known by the resource). Figure 3.12 represents a directed task graph which is part of the application model. **Note:** For reasons of simplification, the communication components are not displayed in this graph. A POOSL specification example of a task component is given in figure 3.13. The described `HandleEvent` method (line 1 to 9) is tail recursive which allows the task component to serve any incoming event. Each incoming event is served by the `Execute` method, which sends (as a message) an execution request to the architecture components. The parameters involved in the computation behaviour are also passed through in the message. When the request of a task execution is granted and served at the architecture level (as

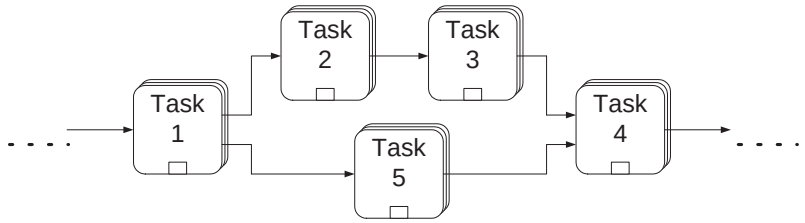


Figure 3.12: Application model represented as a directed task graph

```
        << process >>
Task
<< instantiation parameters >>
TaskName :  String;
ComputationLoad :  Real;
Priority :  Integer;
   << instance variables >>
TaskId :  Integer;
        << methods >>
HandleEvent()()
Execute()()
   << initial method call >>
HandleEvent()()
        << messages >>
in!event
out?event
task!execute
task?executed
```

```
 1 HandleEvent()()
 2 | E : Event|
 3   in?event(E);
 4   par
 5     HandleEvent()()
 6   and
 7     Execute()();
 8     out!event(E)
 9   rap.

10 Execute()()
11   task!execute(TaskId, ComputationLoad,
12     Priority);
13   task?executed(TaskServed
14       | TaskServed = TaskId).
```

Figure 3.13: POOSL specification of a periodic event stream

described in chapter 4), the task component is returned a message `executed` and will generate a new event for a new task component. At line 13 and 14 the task receives an acknowledgement (`task?executed`) when the computation is executed in the architectural level. `TaskServed` is an identifier which is used to check if the right task is executed.
This specification specifies a task component with one input and one output. Task components with for example multiple inputs and/or output can be specified in the same way. Specifying these kinds of tasks must comply with the message protocol used in this method, which are: the `in?event` and `out!event` channels sends and receives an `Event` data classes; communication with the architecture level is done with the parameters `TaskID`, `ComputationLoad`, `Priority` and `TaskServed` over the `task!execute` and `task?executed` channel.

3.3.2 Communication Tasks

As early described, distributed real-time embedded systems commonly consist of communication links. The communication itself depends on the used link (for example bandwidth) and involves the behaviour of the application. In this method the communication taken place over these links are therefore specified in the tasks and resource components. The advantage of this approach is that it improves the exploration of the design space (explore alternative architectures). In this way communication load can easily be mapped on resource component. Each task in the application model exchanges data through communication components, as shown in figure 3.14. From the application point of view, the communication tasks are not specific about the operation of the hardware, with respect to blocking, non-blocking or bandwidth limitation. These communication tasks hold the

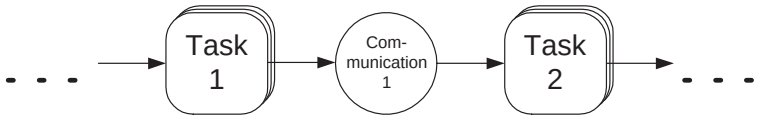


Figure 3.14: Application model consist of application and communication components

communication parameters which involve the application behaviour. Some communication parameters which are specified in the communication tasks are:

Message ID : a unique value used for identification;

Message size : specified in the amount of bytes needed to transfer; other unities are also allowed.

The incoming event in the communication task represents the tail of the message. The outgoing event of the communication task represents the tail of the transferred message through the link, see figure 3.15. The advantage of this

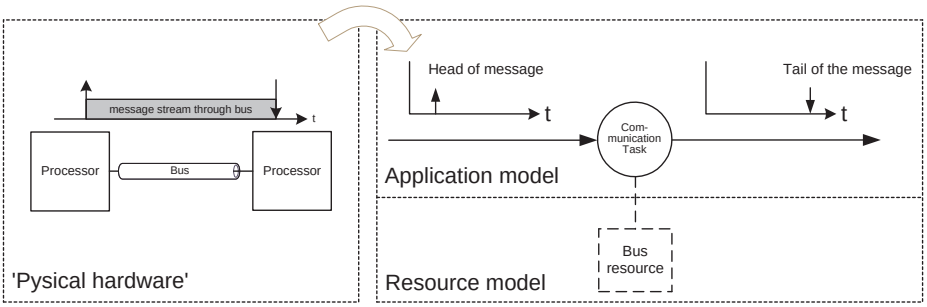


Figure 3.15: Communication interpretation of a distributed real-time embedded system. The communication is specified as a communication task executed on a resource model. The incoming event represent the head of the message and the outgoing event represents the tail of the message.

approach is that processor resource components are not involved with the communication, which simplifies the modelling process. A disadvantage is that this approach models complete buffering of data, which is not always wanted, for example with multimedia streams. A solution would be to transfer long data stream in segments. On the other hand, specifying a communication bridge takes less effort (an extra software and communication task are needed to be specified). A POOSL specification of a communication task component is given in figure 3.16. The method `HandleEvent` is initially called when the model is executed. This recursive method receives all incoming events (seen as the tail of the message) and will model information exchange on a resource component, see the `TransferMsg` method.

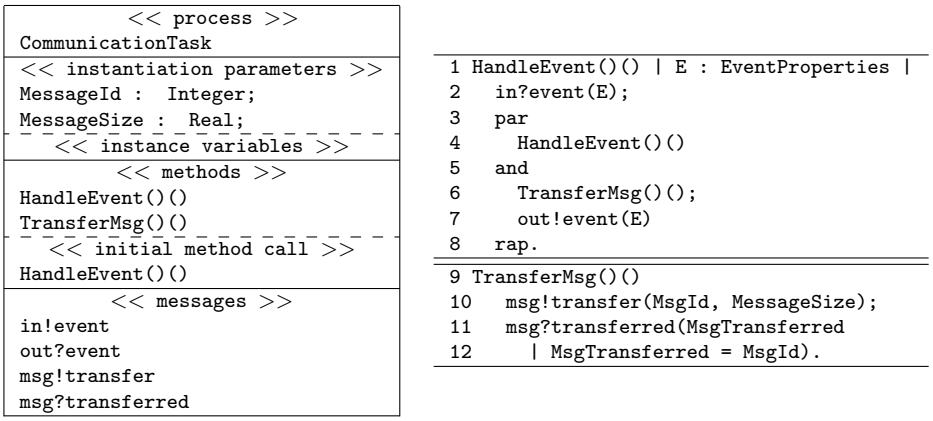


Figure 3.16: POOSL specification of a functional communication component

Chapter 4

Modelling of Architecture characteristics

4.1 Introduction

This chapter describes a method to model the hardware of a real-time embedded system. The hardware model encapsulates the hardware properties. As architectures are so diverse and complex, it is not possible to provide components that cover all possible system architectures. Therefore this chapter presents a set of basic components to model generic hardware structures, which can be used for specification of common (distributed) real-time embedded systems. In this section all physical architectural devices are specified as decoupled resource components as shown in figure 4.1. The interconnections of the architecture are implemented

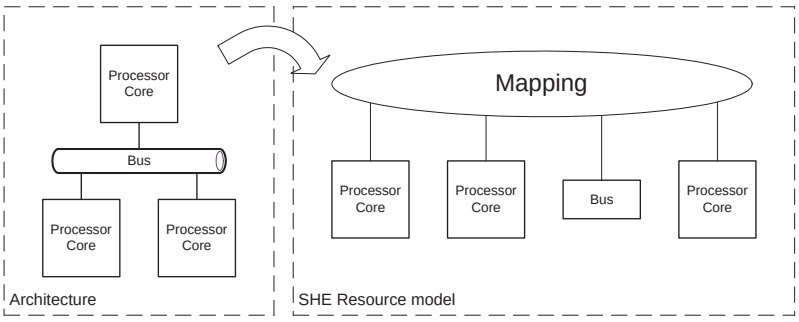


Figure 4.1: Architecture modelled as decoupled SHE components

in the mapping section of this modelling approach. This technique provides a modular modelling approach which allows exploration of alternative architectures. Replacing architecture components is possible without changing the specification of other components. Figure 4.2 shows an example of specifying different architectures on a application model. In this example an architecture which consists of three processors and one shared bus is specified. Figure 4.2 shows an application graph which is mapped on resource components. The channels between the application model and resources model map each task on a resource component. Task 1,2 and 3 are computed on processor core A, B and C respectively. The communication between these task is performed on the bus resource component. A design exploration of a system which consist of two processors and one communication link can easily be established. By removing processor core B and

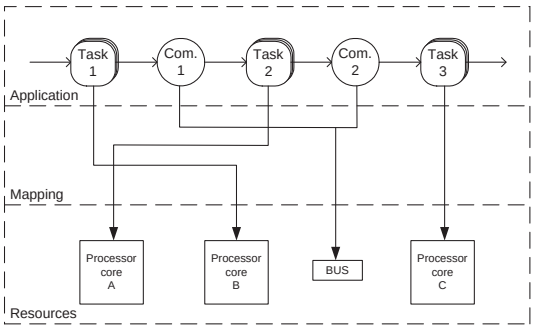


Figure 4.2: An example of a specification of an architecture which consist of three processor cores and one shared bus.

inserting a channel between task 1 and processor core C defines a real-time embedded system which consists of two processors and one shared bus. Figure 4.3 visualises the specification of an application performed on two processor cores and a shared bus.

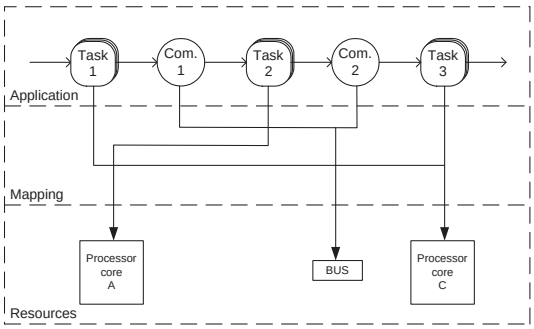


Figure 4.3: An example of a specification of an architecture which consist of two processor cores and one shared bus.

This chapter is organised as follows:

- Section 4.2 describes several components to model the computational part of the hardware in a real-time embedded system.
- Section 4.3 describes how to model communication behaviour of the hardware.

4.2 Modelling of Computation

In this thesis, the processor component models the computation behaviour of the architecture. A simple processor component is given in figure 4.4. This computation component models task executions using a First Come First Served (FCFS) discipline. When a task is received (line 3) the component models a computation using the `delay` procedure (see line 4). When the computation terminates a message is returned to the application model (line 5).

To be able to model basic computation behaviour, it is useful to capture the commonly used scheduling policies. A commonly used scheduling policy is based on an off-line table-driven approach (time-slice scheduling), where the time line is divided into fixed-sized slices. Tasks are statically allocated to slots based on their

```
<< process >>
Computation
<< instantiation parameters >>
MIPS : Real;
<< instance variables >>
<< methods >>
ComputeTask()()
<< initial method call >>
ComputeTask()()
<< messages >>
execute?task
execute!task
```

```
1 ComputeTask()()
2 |Task :  TaskElement, ComputationLoad:Real|
3   execute?task(Task, ComputationLoad);
4   delay ComputationLoad / MIPS;
5   executed!task(Task);
6   ComputeTask()().
```

Figure 4.4: POOSL specification of a computation component without scheduler

rates (periods of execution) and execution requirements. A scheduling approach based on priorities is also commonly used. In this policy a priority is assigned (statically or dynamically) to each task and the execution order is generated on-line based on the current priority value. Two main scheduling algorithms based on priorities are Rate Monotonic (RM) and Earliest Deadline First (EDF). In the RM approach, tasks are assigned with fixed priorities according to their period. The task which needs to be executed at the highest rate receives the highest priority. Once the execution is started, the task can be preempted at any time by a task with a higher priority. With the EDF algorithm priorities are dynamically assigned to tasks, depending on their absolute deadline. EDF is harder to implement but may perform better results, see [22]. The next sections discuss the implementation of a processor model using an off-line table-driven, a RM and an EDF approach.

4.2.1 Static Time Slicing Scheduling

Time slice schedulers assign a time slot to a task for computation. When the computation of the task is not able to be finished in time, the computation will be preempted and placed in a buffer. After this the scheduler activates the next process, see figure 4.5. This technique is comparable to Round Robin (RR) scheduling,

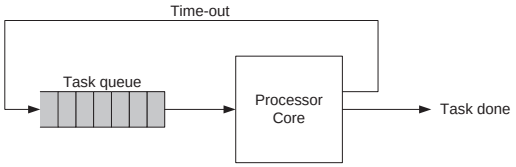


Figure 4.5: Task queue of a time line scheduling algorithm.

see [23]. Both scheduling policies assign a fixed computation time to each task. The RR scheduler is always (during execution) able to accept new tasks in the task queue (as First Come First Serve), where in static time slicing scheduling the order of process execution is fixed and assigned to a time slot by the engineer. The advantage of this approach is that the computation order is fixed when each process is dedicated to a time slot which is large enough to finish the computation. Assigning an execution (task) to multiple slots is allowed. When a process is not able to be finished in the assigned time slot(s), the process will be preempted which consumes time. On the other hand this scheduling approach is not suitable for execution of unknown processes, each task must be assigned to a slot before execution. Figure 4.6 shows the preemptive static time slicing scheduling algorithm specified in POOSL. The `ContextSwitch` and `HandleTaskQueue` methods

```
<< process >>
Computation
<< instantiation parameters >>
MIPS : Real;
SliceTime : Real;
TaskQueue : Dictionary;
<< instance variables >>
<< methods >>
ini()()
ContextSwitch()()
HandleTask()()
HandleTaskQueue()()
<< initial method call >>
ini()()
<< messages >>
task?execute
task?executed
```

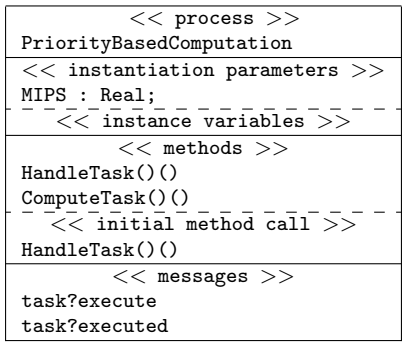
```
1  ini()()
2    par
3      ContextSwitch()();
4    and
5      HandleTaskQueue(1)();
6    rap.

7  ContextSwitch()()
8  | i :  Integer | i := 1;
9    while i <= TaskQueue occupation do
10     activeTask := TaskQueue at(i);
11     delay SliceTime;
12     i := i + 1
13   od;
14   ContextSwitch()().

15 HandleTask(SliceId :  Integer)()
16 |TaskId, SeveringTask :  Integer,
17   ComputationLoad :  Real|
18   SeveringTask := TaskQueue at(SliceId);
19   task?execute(TaskId | TaskId =
20     ServingTask);
21   [activeTask = TaskId]
22     delay ComputationLoad / MIPS;
23   task!executed(TaskId);
24   HandleTask(SliceId)().

25 HandleTaskQueue(SliceId :  Integer)()
26   if TaskQueue occupation > SliceId then
27     par
28       HandleTask(SliceId)()
29     and
30       HandleTaskQueue(SliceId + 1)()
31     rap
32   fi.
```

Figure 4.6: POOSL specification of a preemptive computation resource based on the static time slicing scheduler

described in figure 4.6 at line 3 and 5 are executed in parallel. The tail-recursive `ContextSwitch` method changes the `activeTask` when a predefined time slice period has elapsed. The `activeTask` guard represents the identification of a task that is allowed to be executed. The method `HandleTask` is executed several times depending on the number of scheduled tasks. The most important issue in time slice scheduling is the size of a slice. When the time slice is set small, tasks with a short execution time are finished fast whereas tasks with large execution time are finished late. When task have a short deadline this must be avoided.

4.2.2 Priority Based Scheduling

Most priority based algorithms such as the Rate Monotonic (RM) approach are pre-emptive scheduling policies. This means that a context switch will take place when a task of a higher priority is received. In this method (see section 3.3.1) the priorities are assigned in the task at the application level. The priorities are assigned to tasks before execution and do not change over time. To comply to the RM approach tasks with shorter periods (higher request rates) will have higher priorities. Moreover, the following condition must be met; for every task i ($i = 1, 2, 3, ...$), $C_i < T_i$, where C_i and T_i denotes the computation time and period of task i. To comply with RM scheduling each task must be independent and have a zero offset (for more details see [24]). In figure 4.7, the POOSL specification of a computation resource with priority based scheduling is presented, which also can be used for computation based RM scheduling. Note: For complying to the RM




```
1 HandleTask()()
2   task?execute(Task, ComputationLoad, Priority);
3   Computetask(Task, ComputationLoad, Priority)();
4   HandleTask()().
```

```
5 ComputeTask(ServingTask :  Integer, ServingComputationLoad :  Real,
6       ServingPriority :  Integer)()
7 | ReqTask, ReqPriority :  Integer, ReqComputationLoad :  Real |
8   interrupt delay ServingComputationLoad / MIPS with
9   (
10     task?execute(ReqTask, ReqComputationLoad, ReqPriority |
11               ReqPriority > ServingPriority);
12     ComputeTask(ReqTask, ReqComputationLoad, ReqPriority)()
13   );
14   task!executed(ServingTask).
```

Figure 4.7: POOSL specification of computation with priority scheduling policy

scheduling discipline the priorities must be assigned with respect to their periods.

The initialisation method `HandleTask` is tail-recursive which becomes an endless running process. This procedure guarantees the handling of a computation request of a task. After receiving a task, which has a specific priority and computation load, the `computeTask` method is started. The computation is then preformed on line 8 with the `delay` statement. During the computation this process can be preempted with the `interrupt` statement. The computation will be preempted when a new task is received with higher priority (see line 10 and 11). When a task is preempted the `ComputeTask` method is started again (recursive). When a computation is able to be finished the `ComputeTask` method returns a `task!executed` message to the application level. When the `ComputeTask` is finished the tail-recursive procedure `HandleTask` is restarted.

4.2.3 Earliest Deadline First Scheduling

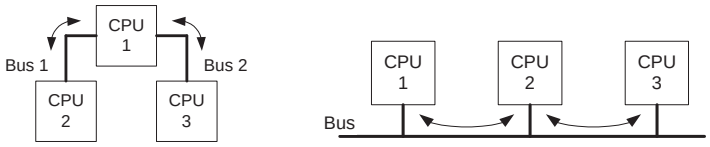
The Earliest Deadline First (EDF) scheduling algorithm dynamically assigns priorities with respect to the absolute deadline of each task. As described in [22], EDF can results in less runtime overhead than RM, when context switches are taken into account. (It is commonly believed that EDF introduces a larger runtime overhead than RM, because in EDF absolute deadlines need to be updated from one task to the other. It is true that this needs extra computation time, but it reduces the costly context switches.) Replacing line 10 and 11 of figure 4.7 by the one given in figure 4.8, result in an EDF based computation model.

```
10 task?execute(ReqTask, ReqComputationLoad, ReqDeadline
11         | ReqDeadline > ServingDeadline);
```

Figure 4.8: This POOSL specification is needed in figure 4.7 to model a earliest deadline first scheduler.

4.3 Modelling of Communication Resources

A communication resource is a facility to exchange data between processors (applications). Today many real-time embedded systems support more functionality by use of multiple processors. To exchange data these systems contain communication resources. [25] discusses several kinds of communication networks used in real-time embedded systems. As existing communication networks are so diverse and complex, it is not possible to provide components that cover all possible communication networks. In general, communication networks can be divided in point-to-point and broadcast networks as shown in figure 4.9. In simple point-



(a) Point-to-point network *(b)* Broadcast network

Figure 4.9: General communication networks

to-point communication network a task sends a message to another one by using a communication resource that has a direct connection between two processors. In common used point-to-point switched networks, where several switched are used, there is no direct connection. In this thesis, a simple example of a point-to-point resource used for a direct connection between two processors is specified. Figure 4.10 specifies a (simple) point-to-point communication component, which models message passing between tasks through the communication resource. This

```
<< process >>
Point2PointCommunication
<< instantiation parameters >>
Bandwidth : Integer;
<< instance variables >>
<< methods >>
HandleMsg()()
TransferingMsg()()
<< initial method call >>
HandleMsg()()
<< messages >>
msg?transfer
msg?transferred
```

```
1 HandleMsg()()
2 | MessageId, MessageSize : Integer |
3   msg?transfer(MessageId, MessageSize);
4   TransferingMsg(MessageSize)();
5   msg!transferred(MessageId);
6   HandleMsg()().
7 TransferingMsg(MessageSize : Integer)()
8   delay MessageSize / Bandwidth;
```

Figure 4.10: POOSL specification of a direct point-to-point communication resource

point-to-point resource exchange data as First Come First Serve (FCFS). Method `HandleMsg` sequentially receives requests through the `msg?transfer` messages from the communication task model. When receiving a communication request, method `TransferingMsg` models a data exchange through the communication resource. The transfer time in this resource depends on the data(`MessageSize`) and bandwidth (see line 8).

Nowadays communication through a broadcasted network (shared medium)

is often used in embedded systems consisting of multiprocessors. A task sends a message through the network to another task running on a different processor. The presented example is able to model the basic behaviour of such networks. For example a central arbiter is neglected. As existing broadcast networks are so diverse and complex it is difficult to model each behaviour and therefore out of the scope of this thesis.

Chapter 5

Mapping

When the application and the architecture models have been defined as described in chapters 3 and 4, the design modelling can be continued using the Y-chart approach (fig 2.2). Mapping is the next section of the Y-chart. In this mapping phase, application components are dedicated to resource components. The task in the application graph will be executed on the resource components. In other words, the workload of the application is assigned to resource components in the architecture model. Figure 5.1 shows the mapping phase of the modelling methodology. This modelling method uses communication channels provided by

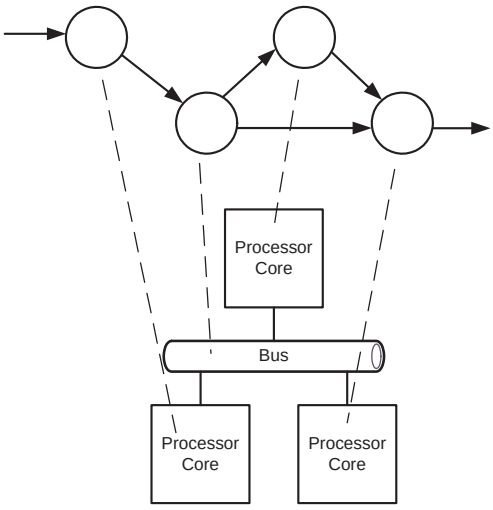


Figure 5.1: Mapping phase.

the POOSL language to map application components to resource components. Figure 5.2 shows an application model mapped on an architecture model specified using the earlier described modelling approach. The figure shows three sections of the Y-chart, namely the application, mapping and architecture(resource models). Creating a mapping channel in SHEsim is done by creating a message channel between the task and the resource components. In this figure (software) task 1 to task 4 and communication task 1 and 2 are defined in the application model. (Software) Task 1 and 2 are mapped on processor A (a resource component). Task 1 will be executed, with respect to the scheduling policy, on processor A when an event has triggered the application model. When processor A has finished the execution of task 1, it sends a message back to task 1 (consuming no time).

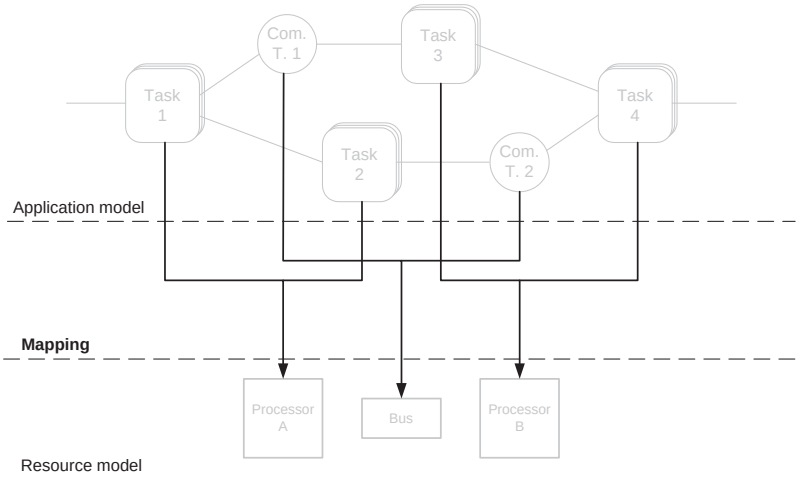
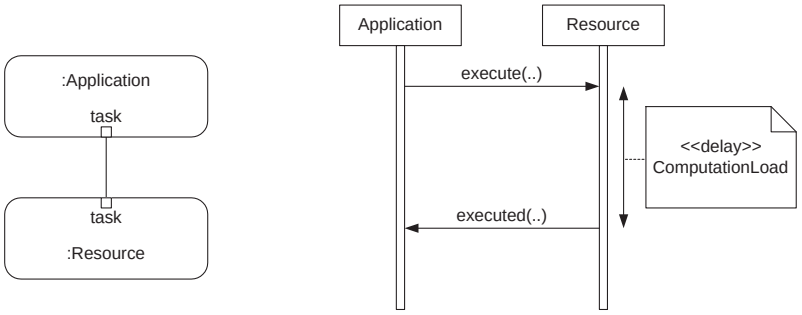


Figure 5.2: Mapping of application and architecture models where the communication channels defines the hardware structure of the system.

When task 1 receives this message, it triggers task 2 and communication task 1. Task 2 will then be executed on processor A and communication task 1 on the bus resource. This process will continue till task 4 is executed on processor B. The outgoing event at task 4 can be used for analysis purposes. Figure 5.3 shows a possible sequence diagram of the communication which takes place between the application and the resource components. This sequence diagram visualises



Application	Architecture
...	...
`task!execute(TaskId, ComputationLoad, priority);`	`task?execute(TaskId, ComputationLoad, Priority);`
`task?executed(TaskServed | TaskServed = TaskId);`	`delay ComputationLoad`
...	`task!executed(TaskId)`
	...

Figure 5.3: Sequence diagram of communication for execution demand in the mapping phase.

the communication between a task in the application level and a processor in the resource level. A task component sends the message `task!execute` to a resource component. The resource component receives this message with a task identification, computation load and a priority. This exchange of data consumes no time. The resource component will execute (with respect to a scheduling policy) the computation load with the `delay` statement. This execution will consume time. When the execution is finished the resource component sends a `task!executed` with the task identification back. The application will only accept messages where the identification is identical of the original. This must be

checked because multiple tasks can be mapped on one resource.

A possible extension of this work is to define a dynamic mapping approach. This is possible because this model is specified modular. The dynamic mapping approach must use the same message protocol used between the application components and resource components. This extension is out of the scope of this thesis and therefore proposed as future work.

Chapter 6

A case study: Distributed In-car radio navigation system

6.1 Introduction

This chapter describes the application of the modelling approach proposed though a case study. This case study, an in-car distributed radio navigation system, is presented in [26]. This is a realistic and well defined system and therefore interesting for performance analysis. In [26] the system is evaluated using Modular Performance Analysis (MPA). MPA is an alternative, analytical performance analysis approach based on the Real-Time Calculus developed at ETH Zurich.

The first section gives a description of the case study. Next the POOSL implementation using the modelling approach defined of chapter 3 to chapter 5. Section 6.3 presents the performance results obtained from the POOSL model. The MPA analysis of the case study is discussed in section 6.4 and followed with a comparison of the POOSL and MPA results (section 6.5). The final section discusses average performance analysis.

6.2 Distributed In-Car Radio Navigation System

The case study presented in [26] is inspired by a system architecture definition for a distributed in-car navigation system. An overview of the system is presented in figure 6.1. It is composed of three main clusters of functionality:

- The Man-Machine Interface (MMI) which takes care of all interaction with the user, such as handling key inputs and graphical display output.
- The navigation functionality (NAV) which is responsible for destination entry, route planning and turn-by-turn route guidance giving the driver both audible and visual advices. The navigation functionality relies on the availability of a map database, typically stored on a CD or DVD, and positioning information, e.g. speed and Global Positioning System (GPS). The latter is not shown here.
- The radio functionality (RAD) which is responsible for basic tuner and volume control as well as handling of traffic information services such as Radio Data System (RDS) / Traffic Message Channel (TMC). RDS TMC is broadcast along with the audio signal of radio channel.

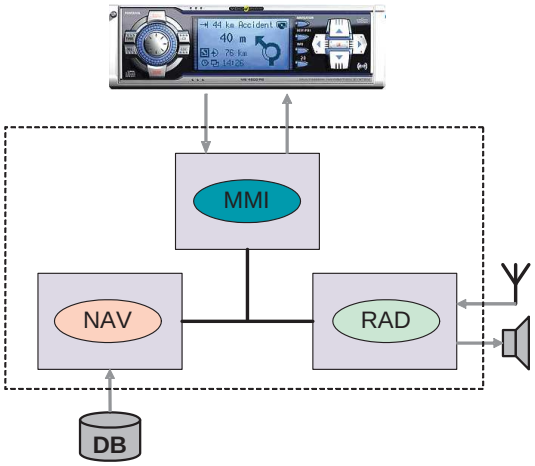


Figure 6.1: High-level of a distributed radio navigation system

The key question that is investigated in [26] is how to distribute the functionality over the available resources, such that we meet our global timing requirements. The functionality is specified with Use-Cases and their associated sequence diagrams. The three selected distinctive scenarios that are used for performance analysis are:

1. "Change Volume" - The user turns the rotary button and expects instantaneous audible feedback from the system. Furthermore, the visual feedback (volume setting on the screen) should be timely and synchronised with the audible feedback. This seemingly trivial Use-Case is actually quite complex because many components are affected. Changing volume might involve commanding a digital signal processor (DSP) and an amplifier in such a way that the quality of the audio signal is maintained while changing the volume. This scenario is shown in detail in figure 6.2. Note that three operations are identified, *HandleKeyPress*, *AdjustVolume* and *UpdateScreen*. Execution times, event rates and message sizes are estimated and annotated in the Sequence Diagram together with the main timing requirements applicable to this scenario.

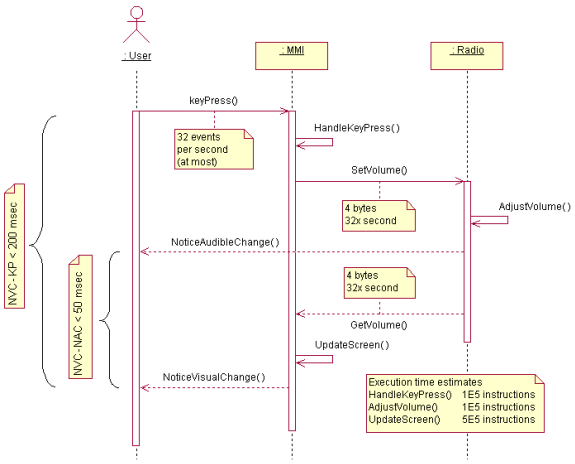


Figure 6.2: Annotated Sequence Diagram for "Change Volume"

2. "Address Look-up" - The destination entry is supported by a smart "typewriter" style interface. By turning a knob the user can move from letter to letter. The map database is searched for each letter that is selected and only those letters in the on-screen alphabet are enabled that are potential next letters in the list. This scenario is shown in detail in figure 6.3. Note that the *DatabaseLookup* operation is expensive compared to the other operations and that the size of the output value the operations and that the size of the output value of operation is 16 times larger than the input message.

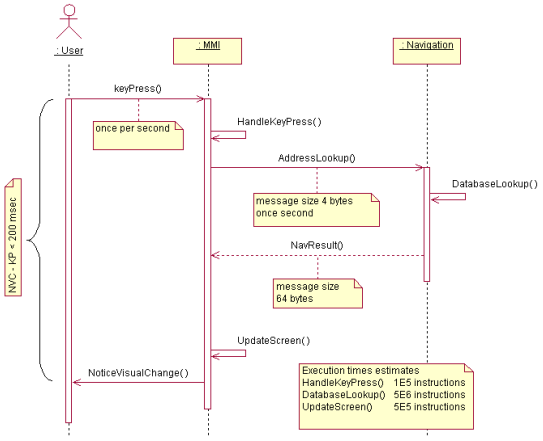


Figure 6.3: Annotated Sequence Diagram for "Address Look-up"

3. "TMC Message Handling" - Digital traffic information is very important for in-car radio navigation systems. It enables features such as automatic replanning of the planned route in case a traffic jam occurs ahead. It is also increasingly important to enhance road safety by warning the driver, for example when a ghost driver is spotted on the planned route. RDS RMC is such a digital traffic information service. TMC messages are broadcast by radio stations together with stereo audio sound. RDS TMC message types are transmitted. The map database is accessed to translate these identifiers and to construct human readable text. The TMC message handling scenario is shown in figure 6.4.

The above presented scenarios can occur in parallel, which means that the system receives TMC messages while a user is pressing the rotary knob. The architectures shown in figure 6.1 suggest to assign the three clusters of functionality each to its own processing unit. Figure 6.5 propose more potential architectures that might be applicable. Those architectures are specified from datasheets of several commercially available automotive CPUs.

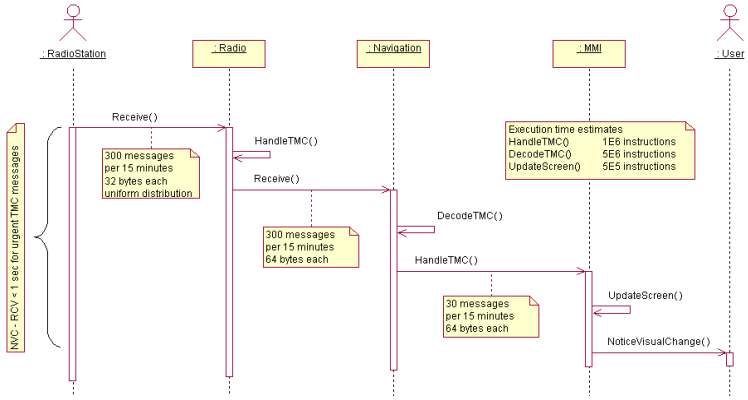


Figure 6.4: Annotated Sequence Diagram for "TMC Message Handling"

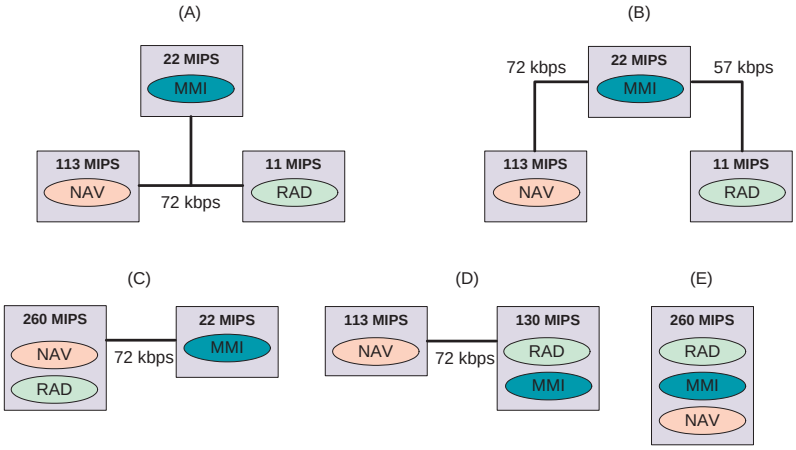


Figure 6.5: Alternative system architecture to explore

6.3 POOSL Model

The SHESim tool, which supports modelling and specification of complex concurrent systems in accordance with the SHE methodology, is used for modelling the distributed in-car radio navigation system. The approach described in chapter 2 is followed to model the in-car radio navigation system. Figure 6.6 shows the model of the in-car navigation on architecture A (see figure 6.5 for the structure of architecture A). As shown in the figure, the `KnobVol`, `KnopAddr` and `Radio` are models of input devices which trigger the application of the distributed in-car navigation system. The triggering is done by events which are messages in POOSL. The `KnopVol` and `KnopAddr` component generate event streams which models volume change and inserting an address by the user respectively. The `Radio` component generates events which models TMC messages coming from a radio station. These components are specified as described in section 3.2. The `Speaker` and `Display` component are models of output devices. These components register events (messages) produced by the application. This registration is used for analysis purposes. These components are specified as a consuming event component described in section 3.2.5. The components in the centre of the figure represent the application of the system. The components are task processes or communication processes as specified in 3.3. The task components and com-

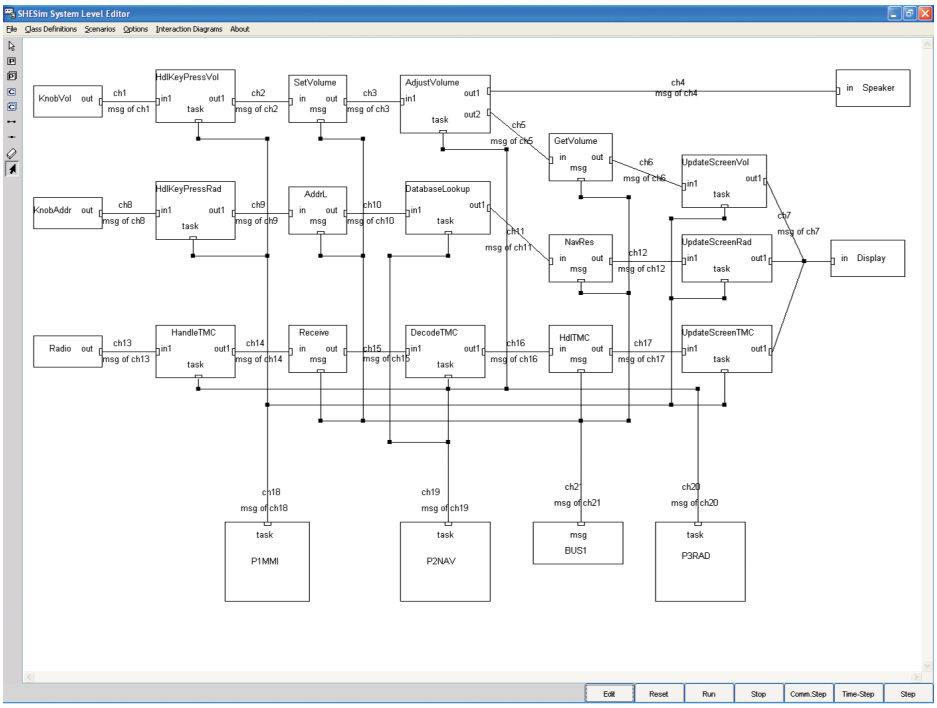


Figure 6.6: SHEsim model of architecture A

munication components are mapped on a processor component (`P1MMI`, `P2NAV` or `P3RAD`) or on a communication component (`BUS1`). The mapping is done by using POOSL communication channels (see chapter 5 for more information). The processor resources are implemented with a priority scheduler as described in section 4.2.2. The resource components[1] do not have a connection between other resources.

6.3.1 Worst Case Performance Analysis

The model described in the previous section is used for analysing the performance of the system. The models of the input devices generate events (messages) periodically as specified in the sequence diagrams in figures 6.2, 6.3 and 6.4. The system performance is analysed with all possible architectures when each scenario is executed individually and when scenarios "Change Volume" or "Address Lookup" are executed in parallel with the "TMC Message Handling" scenario[2]. Note that "Change Volume" and "Address Lookup" are generated from the same knob which means that these scenarios cannot be executed in the same time. The execution of the model is done with the high-speed execution engine Rotalumis which improves the execution speed by a factor of 100. The execution is stopped when no higher WCET is received in half an hour. The results of this analysis are presented in appendix A. Table 6.1 shows the performance numbers of system executed on architecture A. The performance numbers presented in this table and chapter are obtained from the in-car navigation system with architecture `A`. Tasks executed on this architecture are more distributed then other architectures (with

[1] In this chapter the following abbreviation of resources functionalities are used: MMI = Man-Machine Interface, NAV = Navigation, and RAD = Radio.

[2] For clarity the following abbreviations are used: VOL = "Change Volume" scenario, ADDR = "Address Lookup" scenario and TMC = "TMC Message Handling" scenario.

Table 6.1: POOSL performance results of architecture A

Measured scenario	Active scenarios	Worst case delay [ms]	Slack [ms]	Idle time [%] CPU(MMI)	CPU(NAV)	CPU(RAD)
VOL	VOL	41.80	158.20	12.73	100.00	70.91
ADDR	ADDR	79.08	120.92	97.27	95.58	100.00
TMC	TMC	249.20	750.80	99.24	98.53	96.97
VOL	VOL and TMC	75.72	124.28	11.98	98.53	67.88
TMC	VOL and TMC	266.94	733.06	11.98	98.53	67.88
ADDR	ADDR and TMC	86.19	113.81	96.52	94.10	98.77
TMC	ADDR and TMC	244.26	755.74	96.52	94.10	98.77

an exception of architecture B) and it uses a shared communication link. These properties deliver the most interesting performance results. The tables in appendix A shown that all possible architectures meet the application requirements (the requirements (deadlines) are given in section 6.2). The performance analysis showed that for all architectures the slack time is high and that most processors have a long idle time. For example for architecture A, the minimal idle time for the NAV processor is 94.10%. Further analysis of the performance numbers visualises that the obtain worst case delays occurring sporadically. This observation can be shown in a graph, where the horizontal axis denote the measured delay of the event and the vertical axis the occurrence. An example of such a graph is given in figure 6.7 where the ADDR delays are obtained when ADDR and TMC are executed in parallel on architecture A. The circle in the graph display that

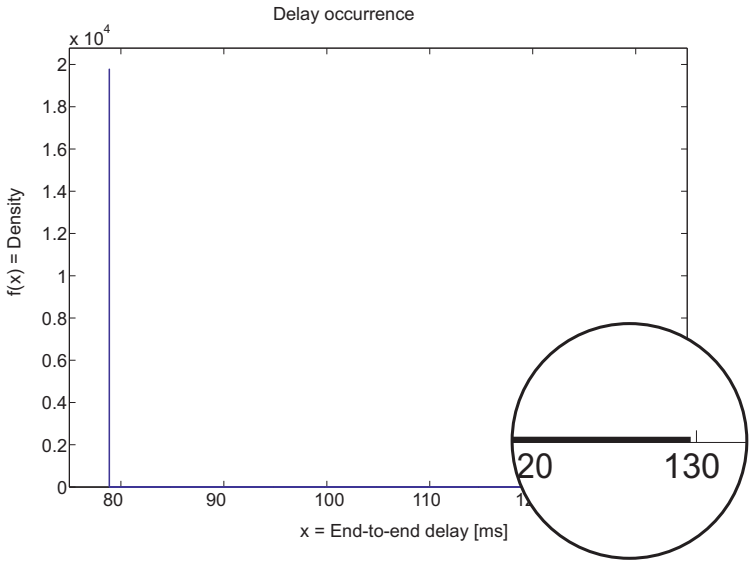


Figure 6.7: Occurrence of ADDR delays when ADDR and TMC are executed in parallel on architecture A

the worst case delay occurs sporadically (the occurrence of a delay is presented with a bar, which accumulates when a delay is in the bounds of the bar).
A performance analysis is also done in [26]. In [26] the system is evaluated using Modular Performance analysis (MPA). MPA is an alternative, analytical performance analysis approach based on the Real-time Calculus developed at ETH Zurich. To make a comparison of the performance values obtained form the POOSL model and MPA method the following section gives a brief description of the MPA method.

6.4 Modular Performance Analyse

Modular Performance Analyse (MPA) uses performance components as basic building blocks to construct a performance model. They define how application tasks are executed on architectural elements and they are the basis for analysis. MPA describe and analyse such a component using real-time calculus. Such a component is given in figure 6.8. An incoming event stream, represented as a

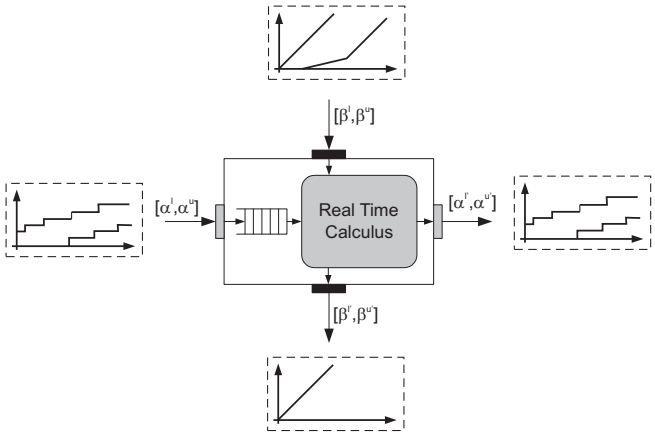


Figure 6.8: A basic performance component with abstract models as input and output and Real-Time Calculus to process internal transformations.

set of upper and lower arrival curves are offered to a FIFO buffer in front of the performance component. The component is triggered by these events and will process them while being restricted by the availability of resources, which are a set of upper and lower services curves. On its output, the component generates an outgoing event stream, represented as a set of upper and lower arrival curves. Resources that are not consumed by the component will be made available again on the resource output of the performance component, again represented as a set of upper and lower service curves. These components are described and analysed using Real-Time Calculus, see [27]. A performance component often computes the convolution and deconvolution defined in min-plus and max-plus calculus. The min-plus convolution and deconvolution definitions are given in appendix A. The performance component uses the following set of equations that describes the processing of abstract event streams and resources:

$$\alpha^{u'} = \min\{(\alpha^u \otimes \beta^u) \oslash \beta^l, \beta^u\} \tag{6.4.1}$$

$$\alpha^{l'} = \min\{(\alpha^l \oslash \beta^u) \otimes \beta^l, \beta^l\} \tag{6.4.2}$$

$$\beta^{u'} = (\beta^u - \alpha^l)\overline{\oslash}0 \tag{6.4.3}$$

$$\beta^{l'} = (\beta^l - \alpha^u)\overline{\otimes}0 \tag{6.4.4}$$

For an extensive discussion about these formulas see [28], [27] and [29]. Performance components can be connected into a network according to the model of a system architecture. Event flows that exit performance components from an event flow output can be connected to an event flow input of another performance component; this will result in horizontal connections. Similarly, resource capacity that is not consumed by a performance component and exits from a resource output can be connected to a resource input of another component; this will result in vertical connections. Together with the models of system resources, i.e.,

the service curves, and with the incoming event streams from the environment, i.e., the arrival curves, can obtain a performance model of a complete system that can be used for performance analysis. An example of specifying the in-car navigation system with architecture A is given in figure 6.9. For better under-

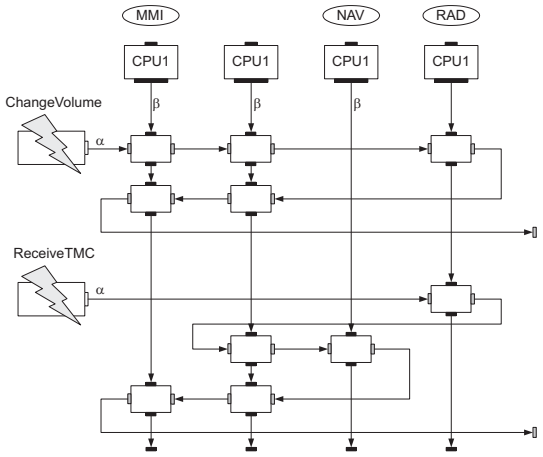


Figure 6.9: MPA model for system architecture A of figure 6.5

standing, this performance method is applied on an example where two event streams are generated in parallel. These two event streams are executed on one resource (processor). The construction of the MPA performance model is given in figure 6.10 The two independent (strictly periodic) event streams `A` and `B` are

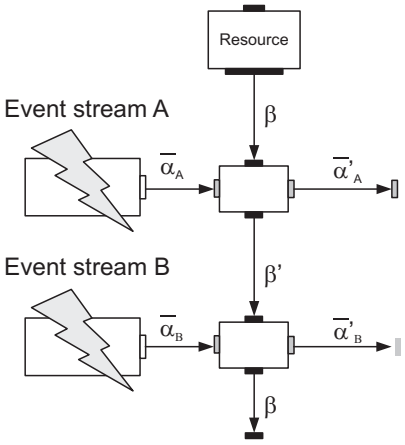
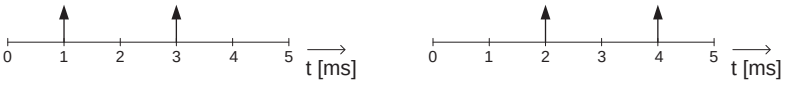


Figure 6.10: MPA model of two event streams sharing one resource.

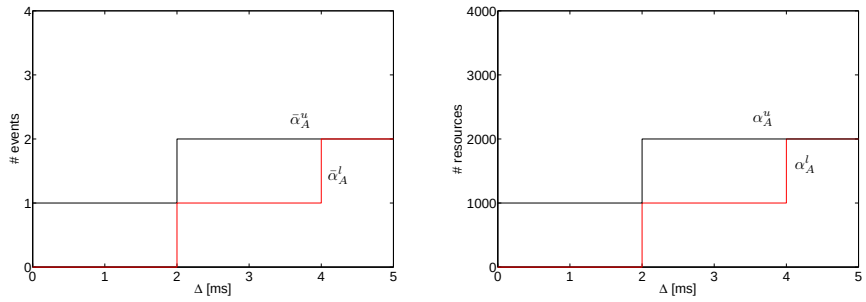
depicted in figure 6.11a and 6.11b respectively. The figure represents the number of events in time. It is assumed that each event requires 1000 resource cycles (the computation). The MPA performance model describes event streams in terms



(a) Event stream A *(b)* Event stream B

Figure 6.11: Event stream A and B in milliseconds

of the minimum and maximum number of events that arrive in a certain time interval (for more details see [27]). From figure 6.11a the arrival curve of event stream `A` is derived.
Figure 6.12a shows the representation of the number of events against a time window of size Δ. Δ denotes the size of time (windows) when events occur between 0 and 5 milliseconds. The figure shows the minimum ($\overline{\alpha}^l_A$) and maximum ($\overline{\alpha}^u_A$) number of events that occur in time window (Δ). Figure 6.12b shows the

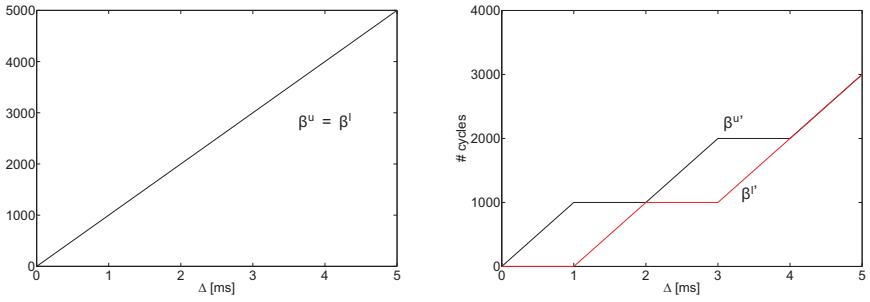


(a) Arrival curve of stream A

(b) Arrival curve against the number of resources of stream A

Figure 6.12: Arrival curves of Stream A, (a) number of events against Δ, (b) number of resource against Δ

representation of the number of resources (minimum and maximum) against Δ. The α^u_A curve represents the upper bound of the required resources (cycles) and α^l_A curve represents the lower bound of the amount of required resources (cycles). The resource service curve of a resource that carries out 1000 cycles each millisecond is given in figure 6.13a. In this case the upper and lower resource service curves are equal. The service curves are linear because of the fixed served cycles of the resource. To calculate the output resource curve β'_A, formulas 6.4.3 and 6.4.4 are used. The result is given in figure 6.13b. From event stream `B`, given in figure

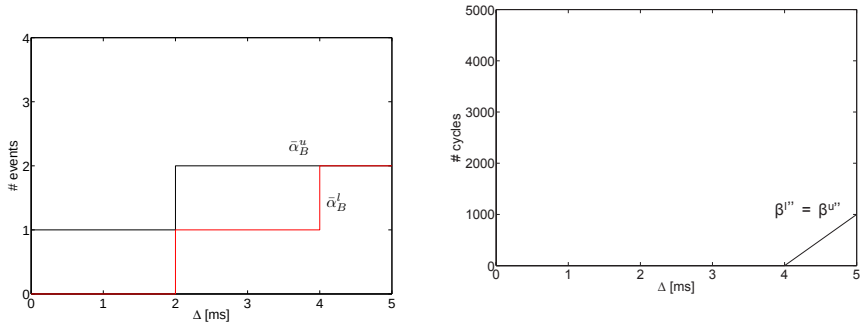


(a) Resource service curve of β (available resources)

(b) Resource service curve of β' (remaining resources)

Figure 6.13: (a) Resource curves of β and β'

6.11b, the arrival curve is derived and shown in figure 6.14. This figure shows the representation of the arrival curve, where the number of events (minimum and maximum) are depicted against time (Δ). When event stream B with arrival curve α_B is processed by the second performance component with service curve β', then the maximum delay d_{max} experienced by event B on the event stream is bounded (horizontal) by the upper event curve and the available resources out of



(a) Arrival curve against the number of resources of stream B

(b) Serve curve of β''

Figure 6.14: Arrival and serve curve of event stream B

the performance component, see d_max in figure 6.15. The maximum buffer space b_{max} that is required to buffer event stream B with arrival curve α_B in the input queue of the second performance component on a resource with service curve β' is bounded (vertical) by the upper event curve and the available resources out of the performance component, see b_max in figure 6.15. Figure 6.15 shows the relations between $\alpha_B^u, \beta_B^{l''}, d_{max}$ and b_{max}. With the chosen event streams and

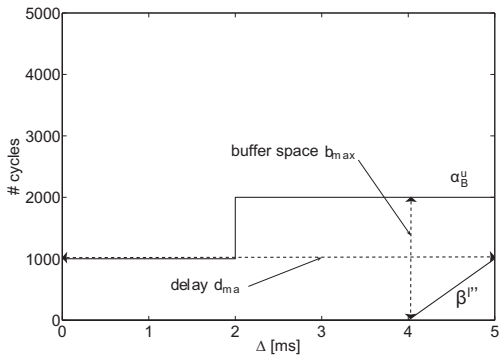


Figure 6.15: Maximum delay and maximum buffer space obtained from arrival and service curves

resource model a delay (time between of releasing and finishing of an event) of only 1 millisecond occurs. The maximum delay derived form the MPA model is 5 milliseconds. This example visualises that MPA can be too conservative. The MPA performance model describes an event stream as a minimum and maximum number of events that arrive in a certain time interval. So these streams do not contain information about when events occur. This is a reason why MPA can result in being too conservative. Creating MPA models is a relatively simple task that require little effort. The advantage of MPA is that the performance calculation is very fast.

6.5 POOSL and MPA Comparison

The case study described in this thesis was used for comparison of different analysis techniques. This and other performance analysis case studies are therefore

made public on [30]. In this section the performance results from the POOSL model are compared with the MPA model. For clarity, this section only describes the comparison of the use case executed on architecture A. The conclusion of this comparison are the same as for the other configurations of the system. A comparison of the worst case performance numbers obtained from the POOSL and MPA analysis is given in table table 6.2. The MPA worst case performance num-

Table 6.2: POOSL and MPA performance results of architecture A

Measured scenario	Active scenarios	Worst case delay [ms]	
		POOSL	MPA
VOL	VOL	41.80	40.91
ADDR	ADDR	79.08	76.07
TMC	TMC	249.20	-
VOL	VOL and TMC	75.72	398.29
TMC	VOL and TMC	266.94	398.29
ADDR	ADDR and TMC	86.19	276.74
TMC	ADDR and TMC	244.26	276.74

bers of scenario TMC were not available at the time of writing this thesis. Note that the MPA analysis only provided the maximum execution time that occur when two scenarios are running in parallel. Noticeable, as expected (MPA can be overconservative because it does not contain information about when events occur) the MPA analysis produces higher worst execution delays when two scenarios are executed in parallel. Comparing the performance numbers obtained when one scenario is executed we see (in the table) that MPA and POOSl are almost equal. The difference between these analysis methods is that MPA is an analytical approach whereas POOSL is based on simulation. This means that POOSL approximates the worst case situation during simulation and MPA derives the upper bound of the worst case execution time. Figure 6.16 shows the domain of these analysis techniques. Figure 6.16 shows a timing diagram where

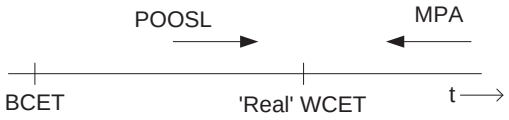


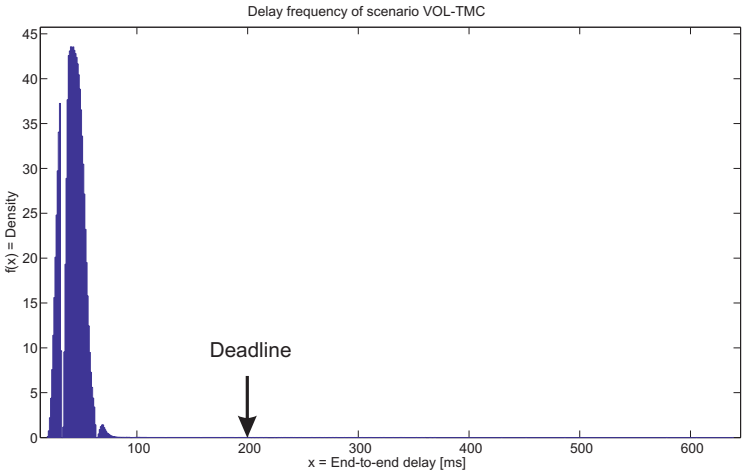
Figure 6.16: Timing diagram which visualise the domain of POOSL and MPA analysis

the best case execution time and the worst case execution time are reflected on a time line. The figure visualises that the POOSL analysis results approximate the 'Real' WCET where the MPA analysis provides upper bound results.

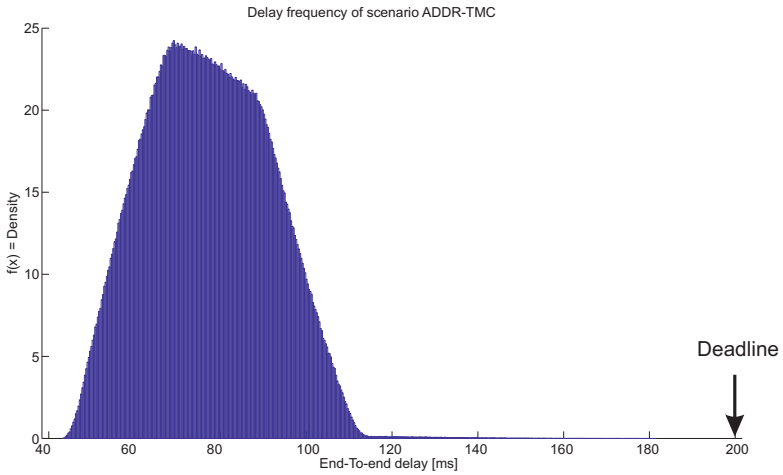
6.6 Average Performance Analysis

In this case study the load of the application is specified in number of instructions. The number of instructions specified in each task denotes the worst case amount needed to execute a task. Using worst case values for performance analysis of hard real-time embedded system is useful. System damage occurs when the system will not fulfil the requirements (e.g deadlines). However, the case study described in this thesis is a soft real-time embedded system, deadline misses will not result in system damage. Therefore performance analysis with worst case values can lead to an over-conservative dimensioning of the system. Specifying the system

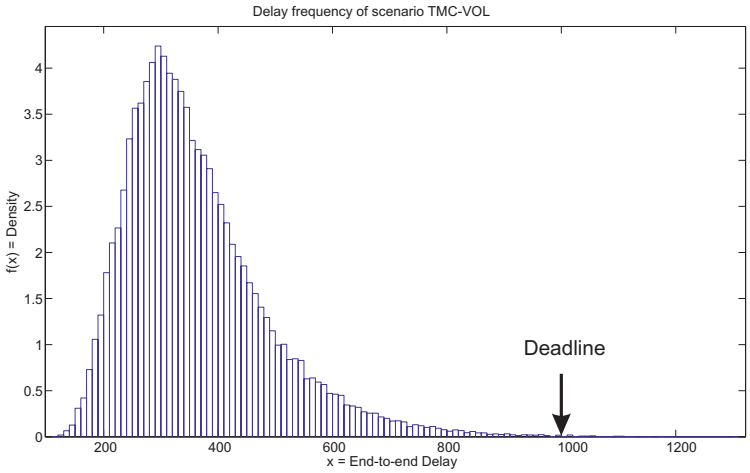
load (number of instructions) between bounds will provide a more realistic performance result.
POOSL is able to specify the load of a system as an distribution. In this thesis the number of instructions is specified as a uniform distribution (Note: Finding a suitable(realistic) load distribution is out of the scope of this thesis and therefore proposed as future work). In this analysis, the number of instruction varies 50% around the worst case value. For example, the amount of instructions of `TaskVolume` (original specified with 1E5 instructions) varies uniform between 5E4 and 15E4 instructions. Figure 6.17 shows the occurrences of the delays when scenarios VOL-TMC, ADDR-TMC, and TMC-VOL are executed in parallel.



(a) scenarios VOL-TMC



(b) scenarios ADDR-TMC



(c) scenarios TMC-VOL

Figure 6.17: Delay frequency functions of scenario VOL, ADDR and TMC

These graphs provide us more insight in the performance of the system. The delay frequency curve shows that most of the delays are close to the end-to-end delay of the scenarios executed separately and obtained by the simulation with fixed load. Table 6.3 presents the average and the worst case delays obtained from the model with uniform load. The results given in table 6.3 shows that the

Table 6.3: Average and WCET obtained from modelling the system with uniform load.

Measured scenario	Average delay [ms]	worst case delay [ms]	Req. deadline [ms]	Deadline misses [%]
VOL-TMC	42.3	638.3	200.0	0.0036
ADDR-TMC	77.2	181.5	200.0	0
TMC-VOL	361.6	1305.7	1000.0	0.0010

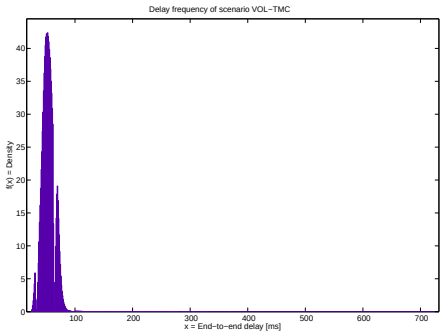
average delays meet the requirements. On the other hand the worst case delay does not meet the requirements, but the deadlines are missed sporadically which is seen from the delay frequency curve and in the right column of the table. Note: these results are obtained from 7.962.000 samples.

6.6.1 Reduction of Resource Performance

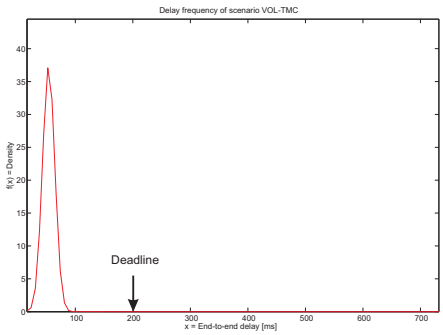
The performance of the architecture will be discussed in this section. The relative low deadline misses and the low processor utilisation shows that this architecture (in this example architecture A) is over-dimensioned. An optimal soft real-time system may have a few deadline misses (depending of the type of system) and high processor occupations. Increasing processor occupation is done by reducing processor speed. The following performance reductions are explored:

- **MMI** - The utilisation of this processor is 88.02%. Simulation has demonstrated that deadlines are already missed. The processor receives VOL events with a period of 32 each second, which means that this processor is loaded with high execution demands. The period and load of the tasks on this processor limits the possibility to reduce the processor speed.
- **NAV** - This processor is involved with the execution of scenario ADDR and TMC. The processor is utilised only for 5.80%. The frequency delay functions of scenario ADDR and TMC (figures 6.17b and 6.17c) show a slack of 80 and 200 milliseconds respectively between the obtained delays and the deadline. Analysis has demonstrated that speed reduction of this processor is feasible without great deadline misses.
- **RAD** - The utilisation of this processor is 32.12%. This processor is involved with the execution of scenario VOL and TMC. Where the VOL scenario has a high rate of execution demands and TMC has a high computation load. The analysis showed a slack time of 100 and 200 milliseconds for ADDR and TMC respectively.

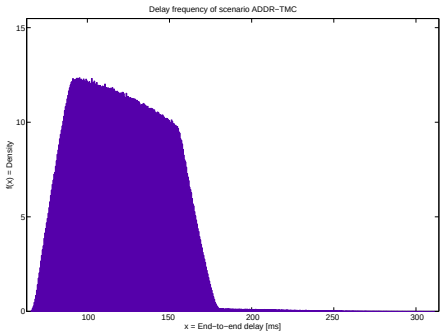
Figures 6.18a, 6.18c and 6.18e show the delay frequency functions of scenarios VOL-TMC, ADDR-TMC and TMC-VOL. These frequency delay are obtained with half of the initial speed of processor 2(NAV) and 3(RAD).



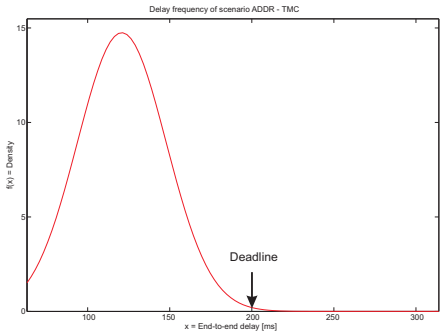
(a) Frequency delay of scenario VOL-TMC



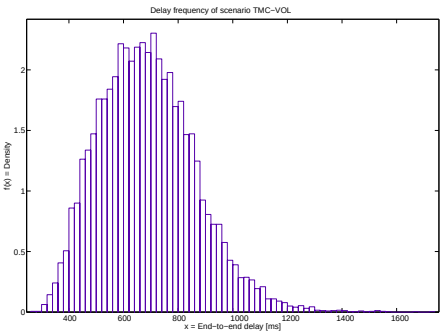
(b) Fitted normal distribution of scenario VOL-TMC



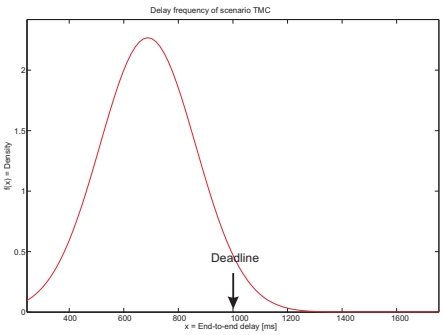
(c) Frequency delay of scenario ADDR-TMC



(d) Fitted normal distribution of scenario ADDR-TMC



(e) Frequency delay of scenario TMC-VOL



(f) Fitted normal distribution of scenario TMC-VOL

Figure 6.18: Frequency delay functions of scenarios VOL-TMC, ADDR-TMC and TMC-VOL when instruction load has a uniform distribution

Table 6.4 shows that the average end-to-end delay meets the requirements (deadlines). The maximum obtained end-to-end delay of the analysis does not meet the requirements. Evaluation of the delay frequency curves show that the deadlines are only missed sporadically.

Table 6.4: Obtained average and maximum delays from the analysis of the POOSL model with a uniform instruction load and reduced resources.

Measured scenario	Average end-to-end delay [ms]	Max. end-to-end delay [ms]	Req. deadline [ms]	Deadline misses [%]
VOL-TMC	53.6	724.7	200.0	0.0077
ADDR-TMC	120.8	311.2	200.0	0.3618
TMC-VOL	688.1	1731.2	1000.0	0.0155

6.6.2 Approximation of Worst Case Performance

The obtained end-to-end delays from the analysis (based on simulation) of the POOSL model does not always find the worst case delays. A distribution curve which fits over the obtained delays provide us values about the worst case delay and the absolute deadline miss. The right graphs of figure 6.18 visualises the end-to-end probability density function fitted on the analysis results. Figure 6.18b, 6.18d and 6.18f are normal distributions. A normal distribution results if the end-to-end delays are the sum of a large number of independent, identically-distributed executions of tasks. This distributions helps to approximate the WCET and to determine the percentage of deadline misses. Table 6.5 shows the probability of

Table 6.5: Calculated deadline misses gathered from the fitted distribution curves

Measured scenario	deadline [ms]	$P(Event > deadline)$ [%]	95% Confidence bound	
			Lower [%]	Upper [%]
VOL-TMC	200.0	$\sim$0.0	-	-
ADDR-TMC	200.0	3.988	3.984	3.993
TMC-VOL	1000.0	5.726	5.464	5.998

deadline misses obtained from the fitted distribution curves. Note that scenario VOL almost never miss the deadline. Applying a distribution curve on obtained performance results (delays) will help to determine the worst case delay of the system. The benefit of this approach is that the simulation time can be reduced and an approximation of the worst case delay can (still) be given.

Chapter 7

Conclusion and recommendations

This chapter describes the realised goals, conclusions and gives recommendations for future developments.

7.1 Realised Objectives and conclusion

In the following enumeration, the numbers relate to the objectives stated in section 1.2.

1. In this thesis we provide a modelling method based on patterns to model and analyse distributed real-time embedded systems. The patterns describe common components of real-time embedded systems like input/output devices, real-time tasks and platform resources. These components are used to specify distributed real-time embedded system model in a modular fashion (plug-and-play). These patterns act as templates that can be applied in other situations by setting the right values of their parameters.

2. The modelling method proposed in this thesis is applied to a realistic case study. The model could be constructed rapidly and in a modular fashion making it suitable for design space exploration. A performance analysis was carried out on this model.

The developed patterns of distributed real-time embedded systems are validated by a performance analysis. The results are compared with an other performance analysis technique (MPA). This comparison shows that the model of the proposed method approximates worst-case values during simulation and MPA derives upper bounds of the worst case execution time. The comparison shows that the proposed method produce performance numbers that approximate the worst case execution as opposed to MPA which is sometimes overly conservative. The proposed method effectively captures the behaviour of both soft and firm real-time embedded systems by use of distributions. As the POOSL analysis technique relies on simulation, the accuracy of the performance results depends on the simulation length. However, we are able to provide deadline miss probabilities by fitting a distribution on the performance analysis results which lead in a less costly platform.

7.2 Recommendations and future research

During the thesis some interesting observations are made to extend the modelling method and to improve the comparison of analysis techniques. The following enumeration describes recommendations and future research.

- In this thesis a method to model distributed real-time embedded systems is described. This method is supported with processor and communication resource components. Specifying memory models increase the modelling of real-time embedded systems domain.
- A static mapping in a modular fashion is applied in this thesis. To decrease the design space exploration time a dynamic mapping approach can be implemented.
- The analysis results are compared based on a single case study. By describing several case studies, each capturing an different analyse problem, a comparison can be made in a systematic way, by considering their pros and cons. Some benchmark issues could be set up based on average performance analysis, worst case performance analysis, the accuracy, the amount of time needed to specify a model, the time needed for performance analysis and the readability of the technique.

Appendix A

Simulation Results of the Distributed In-car Navigation System

A.1 Performance results of architecture A

Scenario	Reaction	End-to-end delay [ms]	Slack [ms]	Occupation [$\neq$]			Idle time [%]				Blocktime [ms]
				P1	P2	P3	P1	P2	P3	BUS	
1	VisualChangeVOL	41.80	158.20	2	0	1	12.73	100.00	70.91	97.16	4.55
1	AudibleChangeVOL	14.08	35.92	2	0	1	12.73	100.00	70.91	97.16	4.55
2	VisualChangeADDR	79.08	120.92	1	1	0	97.27	95.58	100.00	99.24	0.00
3	VisualChangeTMC	249.20	750.80	1	1	1	99.24	98.53	96.97	99.53	77.09
1 and 3	VisualChangeVOL	75.72	124.28	2	1	2	11.98	98.53	67.88	96.68	38.46
1 and 3	AudibleChangeVOL	14.08	35.92	2	1	2	11.98	98.53	67.88	96.68	0.00
1 and 3	VisualChangeTMC	266.94	733.06	2	1	2	11.98	98.53	67.88	96.68	94.83
2 and 3	VisualChangeADDR	86.19	113.81	2	2	1	96.52	94.10	98.77	98.77	7.11
2 and 3	VisualChangeTMC	244.26	755.74	2	2	1	96.52	94.10	98.77	98.77	72.15

A.2 Performance results of architecture B

Scenario	Reaction	End-to-end delay [ms]	Slack [ms]	Occupation [$\neq$]			Idle time [%]					Blocktime [ms]
				P1	P2	P3	P1	P2	P3	BUS1	BUS2	
1	VisualChangeVOL	42.03	157.97	0	1	1	100.00	83.01	70.91	100.00	96.41	0.00
1	AudibleChangeVOL	14.20	35.80	0	1	1	100.00	83.01	70.91	100.00	96.41	0.00
2	VisualChangeADDR	79.08	120.92	1	1	0	77.27	99.47	100.00	99.24	100.00	0.00
3	VisualChangeTMC	270.57	729.43	1	1	1	92.43	99.85	96.97	99.76	99.40	199.77
1 and 3	VisualChangeVOL	70.33	129.67	1	2	2	92.43	82.86	67.85	99.76	95.80	8.8483
1 and 3	AudibleChangeVOL	14.20	35.80	1	2	2	92.43	82.86	67.85	99.76	95.80	0.00
1 and 3	VisualChangeTMC	367.67	632.33	1	2	2	92.43	82.86	67.85	99.76	95.80	47.958
2 and 3	VisualChangeADDR	86.18	113.82	2	2	1	69.69	99.32	96.97	99.01	99.40	6.7056
2 and 3	VisualChangeTMC	270.29	729.71	2	2	1	69.69	99.32	96.97	99.01	99.40	332.67

A.3 Performance results of architecture C

Scenario	Reaction	End-to-end delay [ms]	Slack [ms]	Occupation [#]		Idle time [%]			Blocktime [ms]
				P1	P2	P1	P2	BUS	
1	VisualChangeVOL	28.55	171.45	1	1	98.77	12.73	97.16	0.00
1	AudibleChangeVOL	5.37	44.63	1	1	98.77	12.73	97.16	0.00
2	VisualChangeADDR	54.06	145.94	1	1	98.08	97.27	99.24	0.00
3	VisualChangeTMC	68.30	931.70	2	1	99.17	99.24	99.53	8.27
1 and 3	VisualChangeVOL	62.10	137.90	2	2	98.00	11.97	96.68	33.55
1 and 3	AudibleChangeVOL	5.37	44.63	2	2	98.00	11.97	96.68	0.00
1 and 3	VisualChangeTMC	80.96	919.04	2	2	98.00	11.97	96.68	20.93
2 and 3	VisualChangeADDR	61.04	138.96	2	2	97.31	96.52	98.77	6.98
2 and 3	VisualChangeTMC	101.28	898.72	2	2	97.31	96.52	98.77	41.26

A.4 Performance results of architecture D

Scenario	Reaction	End-to-end delay [ms]	Slack [ms]	Occupation [#]		Idle time [%]			Blocktime [ms]
				P1	P2	P1	P2	BUS	
1	VisualChangeVOL	6.27	193.73	0	1	100.00%	82.77%	100.00%	0.00
1	AudibleChangeVOL	1.98	48.02	0	1	100.00%	82.77%	100.00%	0.00
2	VisualChangeADDR	56.42	143.58	1	1	95.58%	99.54%	99.24%	0.00
3	VisualChangeTMC	70.01	929.99	1	1	98.53%	99.62%	99.53%	0.00
1 and 3	VisualChangeVOL	16.77	183.23	1	2	98.53%	82.38%	96.68%	10.50
1 and 3	AudibleChangeVOL	1.98	48.02	1	2	98.53%	82.38%	96.68%	0.00
1 and 3	VisualChangeTMC	76.16	923.84	1	2	98.53%	82.38%	96.68%	6.15
2 and 3	VisualChangeADDR	63.43	136.57	2	2	94.10%	99.15%	98.77%	7.01
2 and 3	VisualChangeTMC	120.72	879.28	2	2	94.10%	99.15%	98.77%	50.71

A.5 Performance results of architecture E

Scenario	Reaction	End-to-end delay [ms]	Slack [ms]	Occupation [#]	Idle time [%]	Blocktime [ms]
				P1	P1	
1	VisualChangeVOL	2.69	197.31	1	91.38%	0.00
1	AudibleChangeVOL	0.77	49.23	1	91.38%	0.00
2	VisualChangeADDR	21.54	178.46	1	97.85%	0.00
3	VisualChangeTMC	25.00	975.00	2	99.17%	0.00
1 and 3	VisualChangeVOL	4.60	195.40	2	90.55%	1.91
1 and 3	AudibleChangeVOL	0.77	49.23	2	90.55%	0.00
1 and 3	VisualChangeTMC	27.69	972.31	2	90.55%	2.69
2 and 3	VisualChangeADDR	25.38	174.62	3	97.01%	3.85
2 and 3	VisualChangeTMC	6.54	953.46	3	97.01%	21.54

Appendix A

Real-time calculus definitions

In this appendix the definitions of the Min-plus and the Max-plus convolution and deconvolution are given. These definitions are used in the MPA performance components described in section 6.4. For an extensive discussion about these formulas see [28], [27] and [29].
The denoted $\mathcal{F}$ function in the definition refereing to a catalog of functions: Peak rate function , burst-delay function, rate-latency function, affine function, staircase function and step function. For more information see [27].

A.1 Min-plus Convolution and Deconvolution

Definition A.1.1. *[MIN-PLUS CONVOLUTION] Let f and g be two functions or sequences of $\mathcal{F}$. The min-plus convolution of f and g is the function*

$$(f \otimes g)(\Delta) = \inf_{0 \leq \lambda \leq \Delta} \{f(\Delta - \lambda) + g(\lambda)\}$$

If $\Delta < 0$, $(f \otimes g)(\Delta) = 0$.

Similar manner the deconvolution is defined as:

Definition A.1.2. *[MIN-PLUS DECONVOLUTION] Let f and g be two functions or sequences of $\mathcal{F}$. The min-plus deconvolution of f and g is the function*

$$(f \oslash g)(\Delta) = \sup_{\lambda \geq 0} \{f(\Delta + \lambda) - g(\lambda)\}$$

A.2 Max-plus Convolution and Deconvolution

When replacing the infimum (or minimum, it is exists) by a supremum (or maximum, if it exists) similar definition can be derived. For the max-plus convolution $\overline{\otimes}$ and the max-plus deconvolution $\overline{\oslash}$ of two functions f and g are defined as:

Definition A.2.3. *[MAX-PLUS CONVOLUTION] Let f and g be two functions or sequences of $\mathcal{F}$. The max-plus convolution of f and g is the function*

$$(f \overline{\otimes} g)(\Delta) = \sup_{0 \leq \lambda \leq \Delta} \{f(\Delta - \lambda) + g(\lambda)\}$$

If $\Delta < 0$, $(f\overline{\otimes}g)(\Delta) = 0$.

Definition A.2.4. *[MAX-PLUS DECONVELUTION] Let* f *and* g *be two functions or sequences of* $\mathcal{F}$*. The max-plus deconvolution of* f *and* g *is the function*

$$(f\overline{\oslash}g)(\Delta) = \inf_{\lambda \geq 0}\{f(\Delta + \lambda) - g(\lambda)\}$$

References

[1] P.H.A. van der Putten and J.P.M. Voeten. *Specification of reactive hardware/software systems: the method software/hardware engineering (SHE).* PhD thesis, Eindhoven University of Technology, Eindhoven (The Netherlands), 1997.

[2] S.J. Mellor and P.T. Ward. *Structured Development for Real-Time Systems.* Yourdon Press, 1985.

[3] D.J. Hartley and A.I. Pirbhai. *Strategies for Real-Time System Specification.* Dorset House Publishing Co., 1987.

[4] Object management group: Unified model language, www.omg.org/, 2005.

[5] Rational rose realtime. http://www.rational.com/, 2005.

[6] Cinderella sdl 1.3. http://www.cinderella.dk/, 2005.

[7] K.J. Turner. *Using formal description techniques: an introduction to Estelle.* Chichester, 1993.

[8] J. Huang, J.P. M. Voeten, A. Ventevogel, and L. van Bokhoven. Platform-independent design for embedded real-time systems. *Languages for system specification FDL'03*, pages 35–50, 2004.

[9] Z. Huang, J.P.M. Voeten, and B.D. Theelen. Modelling and simulation of a packet switch system using poosl. In *PROGRESS*, 2002.

[10] J. Huang, J. Voeten, P. van der Putten, A. Ventevogel, R. Niesten, and W. van der Maaden. Performance evaluation of complex real-time systems: A case study. In *PROGRESS '02*. STW Technology Foundation, October 2002.

[11] B. D. Theelen, J. P. M. Voeten, and R. D. J. Kramer. Performance modelling of a network processor using poosl. *Comput. Networks*, 41(5):667–684, 2003.

[12] J.P.M. Voeten. Poosl: An object-oriented specification language for the analysis and design of hardware/software systems. EUT 95-E-290, Technische Universiteit Eindhoven, may 1995.

[13] R. Milner. *Communication and Concurrency.* Prentice Hall International Series in Computer Science, 1989.

[14] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pages 526–548, London, UK, 1992. Springer-Verlag.

[15] M.C.W. Geilen, J.P.M. Voeten, P.H.A. van der Putten, L.J. van Bokhoven, and M.P.J. Stevens. Object-oriented modelling and specification using she. *Journal of Computer Languages*, 27(2):19–38, December 2001.

[16] L.J. van Bokhoven. *Constructive Tool Design for Formal Languages: From semantics to Executing Models*. PhD thesis, Eindhiven University of Technology, Eindhoven (The Netherlands), 2002.

[17] B.A.C.J. Kienhuis. *Design Space Exploration of Stream-based Dataflow Architectures: Methods and Tools*. PhD thesis, Delft University of Technology, The Netherlands, January 1999. Explains the Y-chart approach in great detail.

[18] K. Richter, D. Ziegenbein, M. Jersak, and R. Ernst. Model composition for scheduling analysis in platform design. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 287–292, New York, NY, USA, 2002. ACM Press.

[19] L. Thiele and E. Wandeler. Performance analysis of embedded systems. In *The Embedded Systems Handbook*. CRC Press, 2004.

[20] B.D. Theelen. *Performance modelling for system-level design*. PhD thesis, Eindhoven : Technische Universiteit Eindhoven, 2004.

[21] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP 74*, volume IFIP, pages 471–475, 1974.

[22] Giorgio C. Buttazzo. Rate monotonic vs. edf: judgment day. *Real-Time Syst.*, 29(1):5–26, 2005.

[23] William Stallings. *Operating Systems: Internals and Design Principles*. Prentice-Hall inc., third edition, 1998.

[24] Giorgio C. Buttazzo. *Hard real-time computing systems : predictable scheduling algorithms and applications*. Dordrecht : Kluwer Academic Publishers, 1st edition, 1997.

[25] Bhargav P. Upender and Philip J. Koopman. Communication protocols for embedded systems. *Embedded Systems Programming*, 11(7):46–58, November 1994.

[26] E. Wandeler, L. Thiele, M. H. G. Verhoef, and P. Lieverse. System architecture evaluation using modular performance analysis - a case study. In *In 1st International Symposium on Leveraging Applications of Formal Method (ISoLA)*, volume 1, Paphos Cyprus, October 2004.

[27] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[28] Samarjit Chakraborty, Simon Kunzli, and Lothar Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10190, Washington, DC, USA, 2003. IEEE Computer Society.

[29] Martin Naedele Lothar Thiele, Samarjit Chakraborty. Real-time calculus for scheduling hard real-time systems. *International Symposium on Circuits and Systems ISCAS 2000, Geneva, Switzerland*, 4:101–104, March 2000.

[30] http://people.ee.ethz.ch/ leiden05/.